Intensity Frontier
Common Offline Documentation:
art Workbook and Users Guide

Alpha Release 0.70 working draft

July 15, 2014

This version of the documentation is written for version v0_00_18 of the art-workbook code.

Scientific Computing Division
Future Programs and Experiments Department
Scientific Software Infrastructure Group

Principal Author: Rob Kutschke
Editor: Anne Heavey

art Developers: L. Garren, C. Green,
J. Kowalkowski, M. Paterno and P. Russo

List of Chapters

Detailed Table of Contents

List of Figures

List of Tables

List of Code and Output Listings

art Glossary

abstraction	the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details. A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer (adapted from Wikipedia's entry for "Abstraction (computer science)".
analyzer module	an *art* module that may read information from the current event but that may not add information to it; e.g., a module to fill histograms or make printed output
API	Application Programming Interface
art	The *art* framework (*art* is not an acronym) is the software framework developed for common use by the Intensity Frontier experiments to develop their offline code and non-real-time online code
art module	see *module*
art path	a FHiCL sequence of *art* moduleLabels that specifies the work the job will do
artdaq	a toolkit that lives on top of *art* for building high-performance event-building and event-filtering systems; this toolkit is designed to support efficient use of multi-core computers and GPUs. A technical paper on *artdaq* can be found at http://inspirehep.net/record/1229212?ln=en; the *artdaq* home page is at https://cdcvs.fnal.gov/redmine/projects/artdaq/wiki.
bash	a UNIX shell scripting language that is used by some of the support

	scripts in the workbook exercises
boost	a class library with new functionality that is being prototyped for inclusion in future C++ standards
build system	turns source code into object files, puts them into a dynamic library, links them with other libraries, and may also run tests, deploy code to production systems and create some documentation.
buildtool	a Fermilab-developed tool (part of **cetbuildtools**) to compile, link and run tests on the source code of the Workbook
catch	See *exception* in a C++ reference
cetbuildtools	the build system that is used by the *art* Workbook (and by *art* itself).
CETLIB	a utility library used by *art* (developed and maintained by the *art* team) to hold information that does not fit naturally into other libraries
class	The C++ programming language allows programmers to define program-specific data types through the use of *classes*. Classes define types of data structures and the functions that operate on those data structures. Instances of these data types are known as *objects*. Other object oriented languages have similar concepts.
CLHEP	a set of utility classes; the name is an acronym for a Class Library for HEP
collection	
configuration	see *run-time configuration*
const member function	a member function of a class that does not change the value of non-mutable data members; see *mutable data member*
constructor	a function that (a) shares an identifier with its associated class, and (b) initializes the members of an object instantiated from this class
DAQ	data aquisition system
data handling	Broadly, data handling includes "whatever it takes to store data or to get at data." This includes collection, aggregation, access control, sharing,

transfer, bookkeeping, caching, storage, and archiving.

Data Model — see *Event Data Model*

data product — Experiment-defined class that can represent detector signals, reconstructed data, simulated events, etc. In *art*, a *data product* is the smallest unit of information that can be added to or retrieved from an event.

data type — See *type*

declaration (of a class) — the portion of a class that specifies its type, its name, and any data members and/or member functions it has

destructor — a function that (a) has the same identifier as its associated class but prefaced with a tilde ($\sim$), and (b) is used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed

Doxygen — a system of producing reference documentation based on comments in source code

ED — a prefix used in *art* (e.g., for module types) meaning *event-data*

EDAnalyzer — see *analyzer module*

EDFilter — see *filter module*

EDOutput — see *output module*

EDProducer — see *producer module*

EDSource — see *source module*

Event — In HEP there are two notions of the word *event* that are in common use; see *event (unit of information)* or *event (interaction)*. In this documentation suite, unless otherwise indicated, we mean the former.

Event (interaction) — An *event (unit of data)* may contain more than one fundamental interaction; the science goal is always to identify individual fundamental interactions and determine their properties. It is common to use the word *event* to refer to one of the individual fundamental interactions.

In the near detector of a high-intensity neutrino experiment, for example, there may be multiple neutrino interactions within the unit of time that defines a single *event (unit of information)*. Similarly, in a colliding-beam experiment, an *event (unit of information)* corresponds to the information from one beam crossing, during which time there may be multiple collisions between beam particles.

Event (unit of information) In the general HEP sense, an *event* is a set of raw data associated in time, plus any information computed from the raw data; *event* may also refer to a simulated version of same. Within *art*, the representation of an *event (unit of information)* is the classs `art::Event`, which is the smallest unit of information that *art* can process. An `art::Event` contains an event identifier plus an arbitrary number of *data-products*; the information within the *data-products* is intrinsically experiment dependent and is defined by each experiment. For bookkeeping convenience, *art* groups events into a heirarchy: a *run* contains zero or more *subRuns* and a *subRun* contains zero or more events.

Event Data Model (EDM) Representation of the data that an experiment collects, all the derived information, and historical records necessary for reproduction of result

event loop within an *art* job, the set of steps to perform in order to execute the per-event functions for each event that is read in, including steps for begin/end-job, begin/end-run and begin/end-subRun

event-data all of the data products in an experiment's files plus the metadata that accompanies them. The HEP software community has adopted the word *event-data* to refer to the software details of dealing with the information found in *events*, whether the events come from experimental data or simulations.

event-data file a file of event-data, containing either experimental data or simulated events

exception, to throw a mechanism in C++ (and other programming languages) to stop the current execution of a program and transfer control up the call chain; also called *catch*

experiment code — see *user code*

external product — for a given experiment, this is a software product that the experiment's software (within the *art* framework) does not build, but that it uses; e.g., ROOT, Geant4, etc. At Fermilab external products are managed by the in-house UPS/UPD system, and are often called *UPS products* or simply *products*.

FermiGrid — a batch system for submitting jobs that require large amounts of CPU time

FHiCL — Fermilab Hierarchical Configuration Language (pronounced "fickle"), a language developed and maintained by the *art* team at Fermilab to support run-time configuration for several projects, including *art*

FHiCL-CPP — the C++ toolkit used to read FHiCL documents within *art*

filter module — an *art* module that may alter the flow of processing modules within an event; it may add information to the event

framework (*art*) — The *art* framework is an application used to build physics programs by loading physics algorithms, provided as plug-in modules; each experiment or user group may write and manage its own modules. *art* also provides infrastructure for common tasks, such as reading input, writing output, provenance tracking, database access and run-time configuration.

framework (generic) — an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software (significantly abbreviated from Wikipedia's entry for "software framework"); note that the actual functionality provided by any given framework, e.g., *art*, will be tailored to the given needs.

free function — a function without data members; it knows only about agruments passed to it at run time; see *function* and *member function*

Geant4 — a toolkit for the simulation of the passage of particles through matter, developed at CERN. http://geant4.cern.ch/

git	a source code management system used to manage files in the *art* Workbook; similar in concept to the older CVS and SVN, but with enhanced functionality
handle	a type of smart pointer that permits the viewing of information inside a data product but does not allow modification of that information; see *pointer*,*data product*
IF	Intensity Frontier
ifdh_sam	a UPS product that allows *art* to use *SAM* as an external run-time agent that can deliver remote files to local disk space and can copy output files to tape. The first part of the name is an acronym for Intensity Frontier Data Handling.
implementation	the portion of C++ code that specifies the functionality of a declared data type; where as a struct or class declaration (of a data type) usually resides in a *header* file (.h or .hh), the implementation usually resides in a separate source code file (.cc) that "#includes" the header file
instance	see *instantiation*
instantiation	the creation of an object instance of a class in an OOP language; an instantiated object is given a name and created in memory or on disk using the structure described within its class declaration.
jobsub-tools	a UPS product that supplies tools for submitting jobs to the Fermigrid batch system and monitoring them.
Kerberos	a single sign-on, strong authentication system required by Fermilab for access to its computing resources
kinit	a command for obtaining Kerberos credentials that allow access to Fermilab computing resources; see *Kerberos*
member function	(also called *method*) a function that is defined within (is a *member* of) a class; they define the behavior to be exhibited by instances of the associated class at program run time. At run time, member functions have access to data stored in the instance of the class with they are associated, and are thereby able to control or provide access to the state

of the instance.

message facility — a UPS product used by *art* and experiments' code that provides facilities for merging messages with a variety of severity levels, e.g., informational, error, and so on; see also *mf*

message service

method — see *member function*

mf — a namespace that holds classes and functions that make up the message facility used by *art* and by experiments that use *art*; see *message facility*

module — a C++ class that obeys certain rules established by *art* and whose source code file gets compiled into a dynamic library that can be loaded at run-time by *art*. An *art* module "plugs into" a processing stream and performs a specific task on units of data obtained using the Event Data Model, independent of other running modules. See also *moduleLabel*

module_type

moduleLabel — a user-defined identifier whose value is a parameter set that *art* will use to configure a module; see *module* and *parameter set*

Monte Carlo method — a class of computational algorithms that rely on repeated random sampling to obtain numerical results; i.e., by running simulations many times over in order to calculate those same probabilities heuristically just like actually playing and recording your results in a real casino situation: hence the name (Wikipedia)

mutable data member — The

namespace — a container within a file system for a set of identifiers (names); usually grouped by functionality, they are used to keep different subsets of code distinguishable from one another; identical names defined within different namespaces are disambiguated via their namespace prefix

ntuple — an ordered list of *n* elements used to describe objects such as vectors or tables

object — an instantiation of any data type, built-in types (e.g., int, double, float)

or class types; i.e., a location range in memory containing an instantiation

object-oriented language a programming language that supports OOP; this usually means support for classes, including public and private data and functions

object-oriented programming (OOP) a programming language model organized around *objects* rather than procedures, where *objects* are quantities of interest that can be manipulated. (In contrast, programs have been viewed historically as logical procedures that read in data, process the data and produce output.) Objects are defined by *classes* that contain attributes (data fields that describe the objects) and associated procedures. See *C++ class*; *object*.

OOP see *object oriented programming*

output module an *art* module that writes data products to output file(s); it may select a subset of data products in a subset of events; an *art* module contains zero or more output modules

parameter set a C++ class , defined by FHICL-CPP, that is used to hold run-time configuration for *art* itself or for modules and services instantiated by *art*. In a FHiCL file, a parameter set is represented by a FHiCL *table*; see *table*

path a generic word based on the UNIX concept of PATH that refers to a colon-separated list of directories used by *art* when searching for various files (e.g., data input, configuration, and so on)

physics in *art*, *physics* is the label for a portion of the run-time configuration of a job; this portion contains up to five sections, each labeled with a reserved

pointer a variable whose value is the address of (i.e., that points to) a piece of information in memory. A native C++ pointer is often referred to as a *bare pointer*. *art* defines different sorts of *smart pointers* (or *safe pointers*) for use in different circumstances. One commonly used type of smart pointer is called a *handle*.

process_name — a parameter to which the user assigns a mnemonic value identifying the physics content of the associated FHiCL parameter set (i.e., the parameters used in the same FHiCL file). The process_name value is embedded into every data product created via the FHiCL file.

producer module — an *art* module that may read information from the current event and may add information to it

product — See either *external* product or *data* product

redmine — an open source, web-based project management and bug-tracking tool used as a repository for *art* code and related code and documentation

ROOT — an HEP data management and data presentation package supported by CERN; *art*, itself, uses ROOT for persistency of event-data — see ROOT files; user code often also uses ROOT to create histograms, ntuples, trees etc

ROOT files — a file written by ROOT — see ROOT. There are two types of ROOT files managed by *art*: (1) event-data files, and (2) the file managed by TFileService that holds user-defined histograms, ntuples, trees, etc.

run — a period of data collection, defined by the experiment (usually a period of time during which certain running conditions remain unchanged); a run contains zero or more subRuns

run-time configuration — (processing-related) structured documents describing all processing aspects of a single job including the specification of parameters and workflow; in *art* it is supplied by a FHiCL file; see *FHiCL*

safe pointer — see *pointer*

SAM — (Sequential data Access via Metadata) a Fermilab-supplied product that provides the functions of a file catalog, a replica manager and some functions of a batch-oriented workflow manager

scope

sequence (in FHiCL) — one or more comma-separated FHiCL values delimited by square brackets (

...

) in a FHiCL file is called a *sequence* (as distinct from a *table*)

service — in *art*, a singleton-like object (type) whose lifetime and configuration are managed by *art*, and which can by accessed by module code and by other services by requesting a *service handle* to that particular service. The service *type* is used to provide geometrical information, conditions and management of the random number state; it is also used to implement some internal functionality. See also *T File Service*

signature (of a function) — the unique identifier of a C++ a function, which includes: (a) its name, including any class name or namespace components, (b) the number and type of its arguments, (c) whether it is a member function, (d) whether it is a const function (Note that the signature of a function does not include its return type.)

site — As used in the *art* documentation, a *site* is a unique combination of experiment and institution; used to refer to a set of computing resources configured for use by a particular experiment at a particular institution. This means that, for example, the Workbook environment on a Mu2e-owned computer at Fermilab will be different than that on an Mu2e-owned computer at LBL. Also, the Workbook environment on a Mu2e-owned computer at Fermilab will be different from that on an LBNE-owned computer at Fermilab.

smart pointer — see *pointer*

source — (refers to a *data* source) the name of the parameter set inside an FHiCL file describing the first step in the workflow for processing an event; it reads in each event sequentially from a data file or creates an empty event; see also *source code*; see also *EDsource*

source code — code written in C++ (the programming language used with *art*) that requires compilation and linking to create an executable program

source module — an *art* module that can initiate an *art* path by reading in event(s) from a data file or by creating an empty event; it is the first step of the processing chain

standard library, C++ — the C++ standard library, which includes, headers, templates and

	dynamic libraries
std	identifier for the namespace used by the C++ standard library
STL	the C++ Standard Template Library; an archaic name for the C++ Standard Library; see standard library, C++
struct	identical to a C++ class except all members are *public* (instead of *private*) by default
subRun	a period of data collection within a run, defined by the experiment (it may delineate a period of time during which certain run parameters remain unchanged); a SubRun is contained within a *run*; a subRun contains zero or more *events*
table (in FHiCL)	a group of FHiCL definitions delimited by braces ({ ... }) is called a *table*; within *art*, a FHiCL table gets turned into an object called a *parameter set*. Consequently, a FHiCL table is typically called a *parameter set*. See *parameter set*.
template (C++)	Templates are a feature of C++ that allows for meta-programming. In practical terms, the coder can write an algorithm that is independent of type, as long as the type supports the features required by the algorithm. For example, there is a standard library "sort" algorithm that will work for any type that provides a way to determine if one object of the type is "less than" another object of the type.
TFileService	an *art* service used by all experiments to give each module a ROOT subdirectory in which to place its own histograms, TTrees, and so on; see *TTrees* and *ROOT*
truth information	One use of simulated events is to develop, debug and characterize the algorithms used in reconstruction and analysis. To assist in these tasks, the simulation code often creates data products that contain detailed information about the right answers at intermediate stages of reconstruction and analysis; they also write data products that allow the physicist to ask "is this a case in which there is an irreducible background or should I be able to do better?" This information is called the *truth information*, the *Monte Carlo truth* or the *God's block*.

TTrees	a ROOT implementation of a tree; see *tree* and *ROOT*
type	Variables and objects in C++ must be classified into *types*, e.g., built-in types (integer, boolean, float, character, etc.), more complex user-defined classes/structures and typedefs; see *class*, *struct*, and *typedef*. The word *type* in the context of C++ and *art* is the same as *data type* unless otherwise stated.
typedef	A typedef is a different name, or an alias, by which a type can be identified. Type aliases can be used to reduce the length of long or confusing type names, but they are most useful as tools to abstract programs from the underlying types they use (cplusplus.com).
UPS/UPD	a Fermliab-developed system for distributing software products
user code	experiment-specific and/or analysis-specific C++ code that uses the *art* framework; this includes any personal code you write that uses *art*.
variable	a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information, a value. The variable name is the usual way to reference the stored value; this separation of name and content allows the name to be used independently of the exact information it represents.

Part I

Introduction

1 How to Read this Documentation

The *art* document suite, which is currently in an alpha release form, consists of an introductory section and the first few exercises of the Workbook*, plus a glossary and an index. There are also some preliminary (incomplete and unreviewed) portions of the Users Guide included in the compilation.

The Workbook exercises require you to download some code to edit, execute and evaluate. Both the documentation and the code it references are expected to undergo continual development throughout 2013 and 2014. The latest is always available at the *art* Documentation website. Chapter 11 tells you how to keep up-to-date with improvements and additions to the Workbook code and documentation.

1.1 If you are new to HEP Software...

Read Parts I and II (the introductory material and the Workbook) from start to finish. The Workbook is aimed at an audience who is familiar with (although not necessarily expert in) Unix, C++ and Fermilab's UPS product management system, and who understands the basic *art* framework concepts. The introductory chapters prepare the "just starting out" reader in all these areas.

1.2 If you are an HEP Software expert...

Read chapters 1, 2 and 3: this is where key terms and concepts used throughout the *art* document suite get defined. Skip the rest of the introductory material and jump straight

*The Workbook is expected to contain roughly 35 exercises when complete.

into running Exercise 1 in Chapter 9 of the Workbook. Take the approach of: Don't need it? Don't read it.

1.3 If you are somewhere in between...

Read chapters 1, 2 and 3 and skim the remaining introductory material in Part I to glean what you need. Along with the experts, you can take the approach of: Don't need it? Don't read it.

2 Conventions Used in this Documentation

Most of the material in this introduction and in the Workbook is written so that it can be understood by those new to HEP computing; if it is not, please let us know (see Section 3.4)!

2.1 Terms in Glossary

γ The first instance of each term that is defined in the glossary is written in *italics* followed by a γ (Greek letter gamma), e.g., *framework*(γ).

2.2 Typing Commands

Unix commands that you must type are shown in the format `unix command`. Portions of the command for which you must substitute values are surrounded by angle brackets (`< ... >`, e.g., you would type your actual username when you see `<username>`).

While *art* supports OS X as well as flavors of Linux, the instructions for using *art* are nearly identical for all supported systems. When operating-system specific instructions are needed they are noted in the exercises.

When an example Unix command line would overflow the page width, this documentation will use a trailing backslash to indicate that the command is continued on the next line. We indent the second line to make clear that it is not a separate command from the first line. For example:

```
mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/\
  directory1/directory2/directory3
```

You can type the entire command on a single line if it fits, without typing the backslash, *or* on two lines *with* the backslash as the final character of the first line. Do not leave a space before the backslash unless it is required in the command syntax, e.g., before an option, as in

```
mkdir \
  -p <mydir>
```

Computer output from a command is shown as:

```
command output
```

2.3 Procedures to Follow

Step-by-step procedures that the reader is asked to follow are denoted in the following way:

1. First step...
2. Commands inside procedures are denoted as:

   ```
   mkdir -p <mydir>
   ```

2.4 Important Items to Call Out

Occasionally, text will be called out to make sure that you don't miss it. Important or tricky terms and concepts will be marked with an "pointing finger" symbol in the margin, as shown at right.

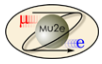
Items that are even trickier will be marked with a "bomb" symbol in the margin, as shown at right. You really want to avoid the problems they describe.

In some places it will be necessary for a paragraph or two to be written for experts. Such paragraphs will be marked with a "dangerous bends" symbol in the margin, as shown at right. Less experienced users can skip these sections on first reading and come back to them at a later time.

2.5 Site-specific Information

Text that refers in particular to Fermilab-specific information is marked with a Fermilab picture, as shown at right.

Text that refers in particular to information about using *art* at non-Fermilab sites is marked with a "generic site" picture, as shown at right. A *site* is defined as a unique combination of experiment and institution, and is used to refer to a set of computing resources configured for use by a particular experiment at a particular institution.



Experiment-specific information will be kept to an absolute minimum; wherever it appears, it will be marked with an experiment-specific icon, e.g., the Mu2e icon at right.

3 Introduction to the *art* Event Processing Framework

3.1 What is *art* and Who Uses it?

art(γ) is an event-processing *framework*(γ) developed and supported by the Fermilab Scientific Computing Division (SCD). The *art* framework is used to build physics programs by loading physics algorithms, provided as plug-in modules. Each experiment or user group may write and manage its own modules. *art* also provides infrastructure for common tasks, such as reading input, writing output, provenance tracking, database access and run-time configuration.

The initial clients of *art* are the Fermilab Intensity Frontier experiments but nothing prevents other experiments from using it as well. The name *art* is always written in *italic lower case*; it is not an acronym.

art is written in C++ and is intended to be used with user code written in C++. (*User code* includes experiment-specific code and any other user-written, non-*art*, non-*external-product*(γ) code.)

art has been designed for use in most places that a typical HEP experiment might require a software framework, including:

- high-level software triggers
- online data monitoring
- calibration
- reconstruction

- analysis
- simulation

art is not designed for use in real-time environments, such as the direct interface with data-collection hardware.

The Fermilab SCD has also developed a related product named *artdaq*(γ), a layer that lives on top of *art* and provides features to support the construction of data-acquisition (*DAQ*(γ)) systems based on commodity servers. Further discussion of *artdaq* is outside the scope of this documentation; for more information consult the *artdaq* home page: https://cdcvs.fnal.gov/redmine/projects/artdaq/wiki.

A technial paper on *artdaq* is available at: http://inspirehep.net/record/1229212?ln=en;

The design of *art* has been informed by the lessons learned by the many High Energy Physics (HEP) experiments that have developed C++ based frameworks over the past 20 years. In particular, it was originally forked from the framework for the CMS experiment, *cmsrun*.

Experiments using *art* are listed at the *art* Documentation website under "Experiments using art."

3.2 Why *art*?

In all previous experiments at Fermilab, and in most previous experiments elsewhere, infrastructure software (i.e., the framework, broadly construed – mostly forms of bookkeeping) has been written in-house by each experiment, and each implementation has been tightly coupled to that experiment's code. This tight coupling has made it difficult to share the framework among experiments, resulting in both great duplication of effort and mixed quality.

art was created as a way to share a single framework across many experiments. In particular, the design of *art* draws a clear boundary between the framework and the user code; the *art* framework (and other aspects of the infrastructure) is developed and maintained by software engineers who are specialists in the field of HEP infrastructure software; this provides a robust, professionaly maintained foundation upon which physicists can develop the code for their experiments. Experiments use *art* as an *external package*. Despite some

constraints that this separation imposes, it has improved the overall quality of the framework and reduced the duplicated effort.

3.3 C++ and C++11

In 2011, the International Standards Committee voted to approve a new standard for C++, called C++ 11.

Much of the existing user code was written prior to the adoption of the C++ 11 standard and has not yet been updated. As you work on your experiment, you are likely to encounter both code written the new way and code written the old way. Therefore, the Workbook will often illustrate both practices.

A very useful compilation of what is new in C++ 11 can be found at

https://cdcvs.fnal.gov/redmine/projects/gm2public/wiki/CPP2011

This reference material is written for advanced C++ users.

3.4 Getting Help

Please send your questions and comments to *art-users@fnal.gov*. More support information is listed at https://web.fnal.gov/project/ArtDoc/SitePages/Support.aspx.

3.5 Overview of the Documentation Suite

When complete, this documentation suite will contain several principal components, or *volumes*: the introduction that you are reading now, a Workbook, a Users Guide, a Reference Manual, a Technical Reference and a Glossary. At the time of writing, drafts exist for the Introduction, the Workbook, the Users Guide and the Glossary. The components in the documentation suite are illustrated in Figure 3.1.

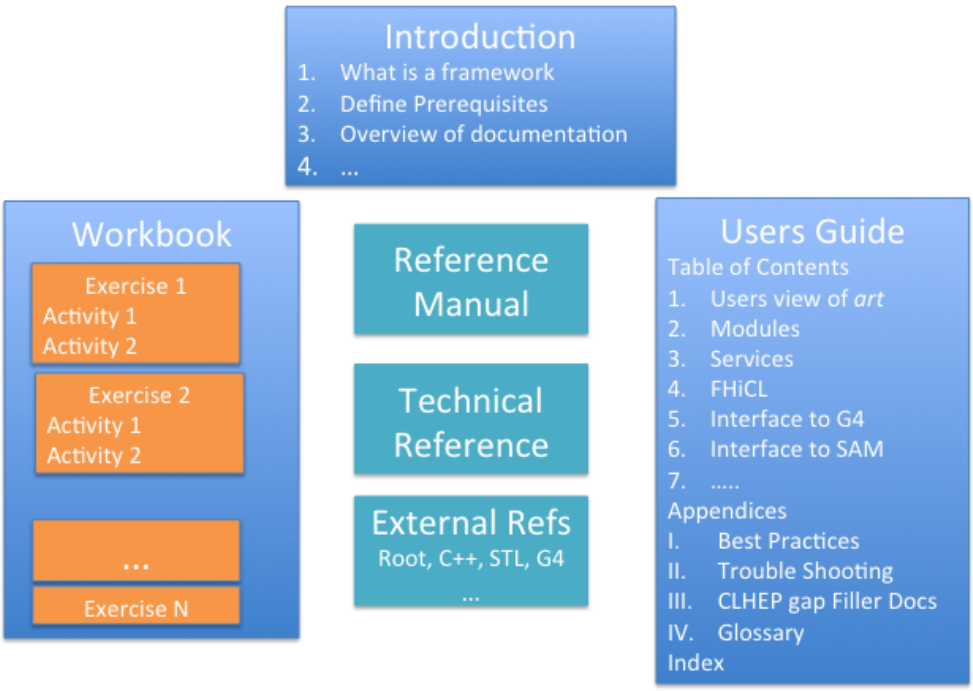


Figure 3.1: Principal components of the *art* documentation suite

3.5.1 The Introduction

This introductory volume is intended to set the stage for using *art*. It introduces *art*, provides background material, describes some of the software tools on which *art* depends, describes its interaction with related software and identifies prerequisites for successfully completing the Workbook exercises.

3.5.2 The Workbook

The Workbook is a series of standalone, self-paced exercises that will introduce the building blocks of the *art* framework and the concepts around which it is built, show practical applications of this framework, and provide references to other portions of the documentation suite as needed. It is targeted towards physicists who are new users of *art*, with the understanding that such users will frequently be new to the field of computing for HEP and to C++.

One of the Workbook's primary functions is training readers how and where to find more extensive documentation on both *art* and external software tools; they will need this information as they move on to develop and use the scientific software for their experiment.

The Workbook assumes some basic computing skills and some basic familiarity with the C++ computing language; Chapter 6 provides a tutorial/refresher for readers who need to improve their C++ skills.

The Workbook is written using recommended best practices that have become current since the adoption of C++ 11 (see Section 3.8).

Because *art* is being used by many experiments, the Workbook exercises are designed around a *toy* experiment that is greatly simplified compared to any actual detector, but it incorporates enough richness to illustrate most of the features of *art*. The goal is to enable the physicists who work through the exercises to translate the lessons learned there into the environment of their own experiments.

3.5.3 Users Guide

The Users Guide is targeted at physicists who have reached an intermediate level of competence with *art* and its underlying tools. It contains detailed descriptions of the features of *art*, as seen by the physicists. The Users Guide will provide references to the *external products*(γ) on which *art* depends, information on how *art* uses these products, and as needed, documentation that is missing from the external products' own documentation.

3.5.4 Reference Manual

The Reference Manual will be targeted at physicists who already understand the major ideas underlying *art* and who need a compact reference to the Application Programmer Interface (*API*(γ)). The Reference Manual will likely be generated from annoted source files, possibly using *Doxygen*(γ).

3.5.5 Technical Reference

The Technical Reference will be targeted at the experts who develop and maintain *art*; few physicists will ever want or need to consult it. It will document the internals of *art* so that a broader group of people can participate in development and maintenance.

3.5.6 Glossary

The glossary will evolve as the documentation set grows. At the time of writing, it includes definitions of *art*-specific terms as well as some HEP, Fermilab, C++ and other relevant computing-related terms used in the Workbook and the Users Guide.

3.6 Some Background Material

This section defines some language and some background material about the *art* framework that you will need to understand before starting the Workbook.

3.6.1 Events and Event IDs

In almost all HEP experiments, the core idea underlying all bookkeeping is the *event*(γ). In a triggered experiment, an event is defined as all of the information associated with a single trigger; in an untriggered, spill-oriented experiment, an event is defined as all of the information associated with a single spill of the beam from the accelerator. Another way of saying this is that an event contains all of the information associated with some time interval, but the precise definition of the time interval changes from one experiment to another *. Typically these time intervals are a few nanoseconds to a few tens of mircoseconds. The information within an event includes both the raw data read from the Data Acquisition System (DAQ) and all information that is derived from that raw data by the reconstruction and analysis algorithms. An event is the smallest unit of data that *art* can process at one time.

In a typical HEP experiment, the trigger or DAQ system assigns an event identifier (event ID) to each event; this ID uniquely identifies each event, satisfying a critical requirement imposed by *art* that each event be uniquely identifable by its event ID. This requirement also applies to simulated events.

The simplest event ID is a monotonically increasing integer. A more common practice is to define a multi-part ID and *art* has chosen to use a three-part ID, including:

- *run*(γ) number
- *subRun*(γ) number
- *event*(γ) number

There are two common methods of using this event ID scheme and *art* allows experiments to chose either:

1. When an experiment takes data, the event number is incremented every event. When some predefined condition occurs, the event number is reset to 1 and the subRun number is incremented, keeping the run number unchanged. This cycle repeats until some other predefined condition occurs, at which time the event number is reset to

*There is a second, distinct, sense in which the word *event* is sometimes used; it is used as a synonym for a *fundamental interaction*; see the glossary entry for *event (fundamental interaction)*(γ). Within this documentation suite, unless otherwise indicated, the word *event* refers to the definition given in the main body of the text.

1, the subRun number is reset to 0 (0 not 1 for historical reasons) and the run number is incremented.

2. The second method is the same as the first except that the event number mononton-ically increases throughout a run and does not reset to 1 on subRun boundaries. The event number does reset to 1 at the start of each run.

art does not define what conditions cause these transitions; those decisions are left to each experiment. Typically experiments will choose to start new runs or new subRuns when one of the following happens: a preset number of events is acquired; a preset time interval expires; a disk file holding the ouptut reaches a preset size; or certain running conditions change.

art requires only that a subRun contain zero or more events and that a run contain zero or more subRuns.

When an experiment takes data, events read from the DAQ are typically written to disk files, with copies made on tape. The events in a single subRun may be spread over sev-eral files; conversely, a single file may contain many runs, each of which contains many subRuns.

3.6.2 *art* Modules and the Event Loop

Users provide executable code to *art* in pieces called *art modules*(γ)[†] that are dynamically loaded as plugins and that operate on event data. The concept of reading events and, in response to each new event, calling the appropriate member functions of each module, is referred to as the *event loop*(γ). The concepts of the *art module* and the *event loop* will be illustrated via the following discussion of how *art* processes a job.

The simplest command to run *art* looks like:

```
art -c <file>.fcl
```

The argument to -c is the *run-time configuration file*(γ), a text file that tells one run of *art* what it should do. Run-time configuration files for *art* are written in the Fermilab

[†]Many programming languagues have an idea named *module*; the use of the term *module* by *art* and in this documentation set is an *art*-specific idea that will be developed through the first few chapters of the Workbook.

Hierarchical Configuration Language *FHiCL*(γ) (pronounced "fickle") and the filenames end in `.fcl`. As you progress through the Workbook, this language and the conventions used in the run-time configuration file will be explained; the full details are available in Chapter 22 of the Users Guide. (The run-time configuration file is often referred to as simply the *configuration file* or even more simply as just the *configuration*(γ).)

When *art* starts up, it reads the configuration file to learn what input files it should read, what user code it should run and what output files it should write. As mentioned above, an experiment's code (including any code written by individual experimenters) is provided in units called *art modules*. A module is simply a C++ class, provided by the experiment or user, that obeys a set of rules defined by *art* and whose *source code*(γ) file gets compiled into a *dynamic library*(γ) that can be loaded at run-time by *art*.

These rules will be explained as you work through the Workbook and they are summarized in *a future chapter in the User's Guide*.

The code base of a typical experiment will contain many C++ classes. Only a small fraction of these will be modules; most of the rest will be ordinary C++ classes that are used within modules[‡].

A user can tell *art* the order in which modules should be run by specifying that order in the configuration file. A user can also tell *art* to determine, on its own, the correct order in which to run modules; the latter option is referred to as *reconstruction on demand*.

Imagine the processing of each event as the assembly of a widget on an assembly line and imagine each module as a worker that needs to perform a set task on each widget. Each worker has a task that must be done on each widget that passes by; in addition some workers may need to do some start-up or close-down jobs. Following this metaphor, *art* requires that each module provide code that will be called once for every event and *art* allows any module to provide code that will be called at the following times:

- at the start of the *art* job
- at the end of the *art* job
- at the start of each run

[‡] *art* defines a few other specialized roles for C++ classes; you will encounter these in Sections 3.6.4 and 3.6.5.

- at the end of each run
- at the start of each SubRun
- at the end of each SubRun

For those of you who are familiar with *inheritance* in C++, a module class (i.e., a "module") must inherit from one of a few different module *base classes*. Each module class must override one pure-virtual member function from the base class and it may override other virtual member functions from the base class.

After *art* completes its initialization phase (intentionally not detailed here), it executes the event loop. This is illustrated in Figure 3.2, which is described in the text below:

1. calls the *constructor*(γ) of every module in the configuration
2. calls the beginJob *member function*(γ) of every module that provides one
3. reads one event from the input source, and for that event
   (a) determines if it is from a run different from that of the previous event (true for first event in loop)
   (b) if so, calls the beginRun member function of each module that provides one
   (c) determines if the event is from a subRun different from that of the previous event (true for first event in loop)
   (d) if so, calls the beginSubRun member function of each module that provides one
   (e) calls each module's (required) per-event member function
4. reads the next event and repeats the above per-event steps until it encounters a new subRun
5. closes out the current subRun by calling the endSubRun member function of each module that provides one
6. repeats steps 4 and 5 until it encounters a new run
7. closes out the current run by calling the endRun member function of each module that provides one

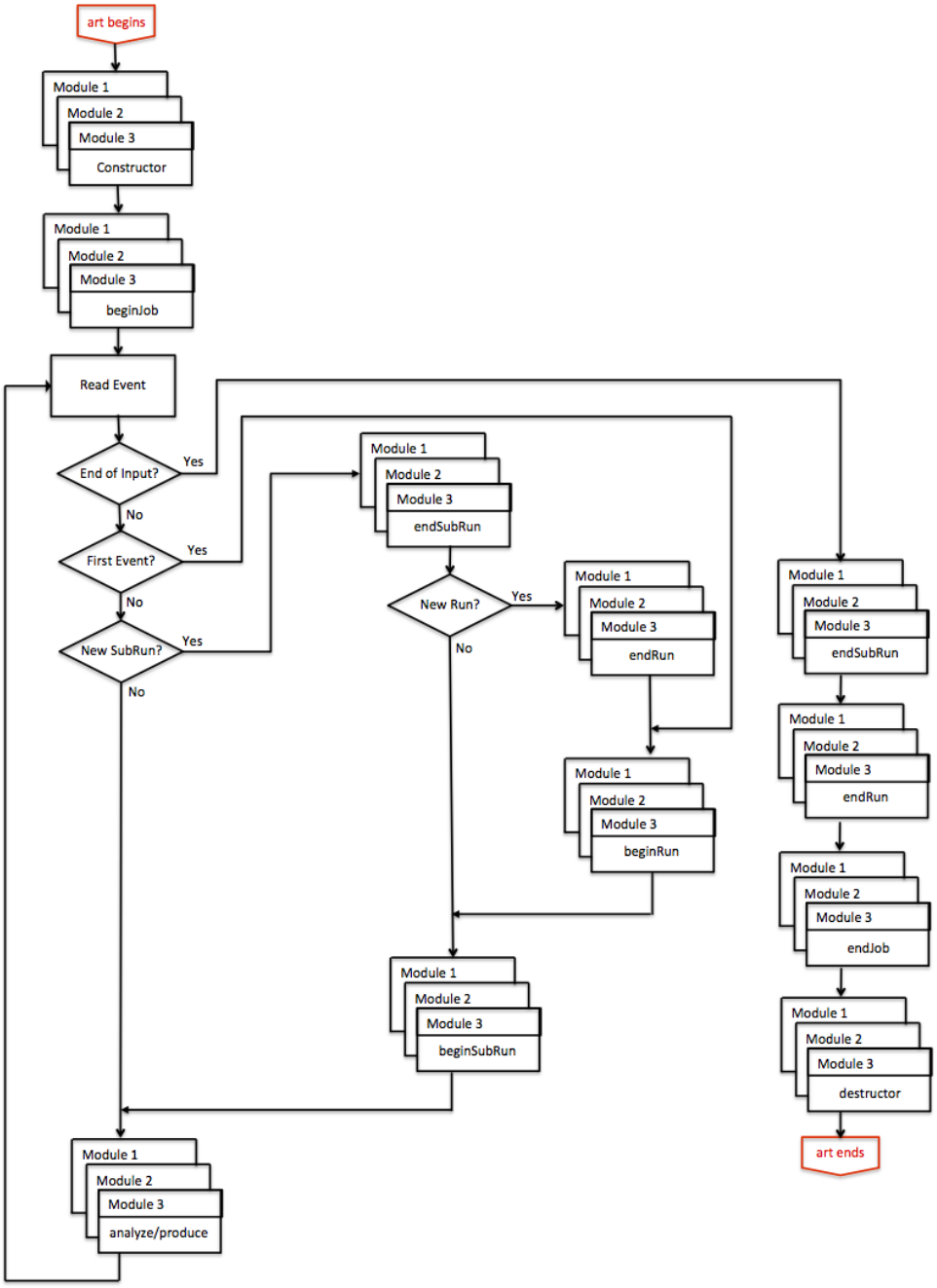


Figure 3.2: Flowchart describing the *art* event loop for an input file that contains at least one event. *art* begins at the box in the upper left and ends at the box in the lower right. On the first event, the tests for new subRun and new run are true. Not all features of the event loop are shown, just those that you will encounter in the early parts of the *art* workbook. The case of a file with no events is not shown because it has many subcases and is not of general interest.

8. repeats steps 3 through 7 until it reaches the end of the input source
9. calls the endJob member function of each module that provides one
10. calls the *destructor*(γ) of each module

This entire set of steps comprises the event loop. One of *art*'s most visible jobs is controlling the event loop.

3.6.3 Module Types

Every *art* module must be one of the following five types, which are defined by the ways in which they interact with each event and with the event loop:

***analyzer module*(γ)** May inspect information found in the event but may not add new information to the event. .

***producer module*(γ)** May inspect information found in the event and may add new information to the event.

***filter module*(γ)** Same functions as a producer module but may also tell *art* to skip the processing of some, or all, modules for the current event; may also control which events are written to which output.

***source module*(γ)** Reads events, one at a time, from some source; *art* requires that every *art* job contain exactly one source module. A source is often a disk file but other options exist and will be described in the Workbook and Users Guide.

***output module*(γ)** Reads selected data products from memory and writes them to an output destination; an *art* job may contain zero or more output modules. An ouptut destination is often a disk file but other options exist and will be described in the Users' Guide. .

Note that no module may change information that is already present in an event.

What does an analyzer do if it may neither alter information in an event nor add to it? Typically it creates printout and it creates ROOT files containing histograms, *trees*(γ) and *nuples*(γ) that can be used for downstream analysis. (If you have not yet encountered these terms, the Workbook will provide explanations as they are introduced.)

Most novice users will only write analyzer modules and filter modules; readers with a little more experience may also write producer modules. The Workbook will provide examples of all three. Few people other than *art* experts and each experiment's software experts will write source or output modules, however, the Workbook will teach you what you need to know about configuring source and output modules.

3.6.4 *art* Data Products

This section introduces more ideas and terms dealing with event information that you will need as you progress through the Workbook.

The term *data product*(γ) is used in *art* to mean the unit of information that user code may add to an event or retrieve from an event. A typical experiment will have the following sorts of data products:

1. The DAQ system will package the raw data into data products, perhaps one or two data products for each major subsystem.
2. Each module in the reconstruction chain will create one or more data products.
3. Some modules in the analysis chain will produce data products; others may just make histograms and write information in non-*art* formats for analysis outside of *art*; they may, for example, write user-defined ROOT `TTree`s.
4. The simulation chain will usually create many data products. Some will be simulated event-data while others will describe the true properties of the simulated event. These data products can be used to study the response of the detector to simulated events; they can also be used to develop, debug and characterize the reconstruction algorithms.

Because these data products are intrinsically experiment-dependent, each experiment defines its own data products. In the Workbook, you will learn about a set of data products designed for use with the toy experiment. There are a small number of data products that are defined by *art* and that hold bookkeeping information; these will be described as you encounter them in the Workbook.

A data product is just a C++ *type*(γ) (a class, *struct*(γ) or typedef) that obeys a set of rules defined by *art*; these rules are very different than the rules that must be followed for a class 

to be a module; when the sections that describe these rules in detail have been prepared, we will add references here. A data product can be a single integer, a large complex class hierarchy, or anything in between.

Add the missing references alluded to in the previous para.

Very often, a data product is a *collection*(γ) of some experiment-defined type. The C++ standard libraries define many sorts of collection types; *art* supports many of these and also provides a custom collection type named `cet::map_vector`. Workbook exercises will clarify the *data product* and *collection type* concepts.

3.6.5 *art* Services

Previous sections of this Introduction have introduced the concept of C++ classes that have to obey a certain set of rules defined by *art*, in particular, modules in Section 3.6.2 and data products in Section 3.6.4. *art services*(γ) are yet other examples of this.

In a typical *art* job, two sorts of information need to be shared among the modules. The first sort is stored in the data products themselves and is passed from module to module via the event. The second sort is not associated with each event, but rather is valid for some aggregation of events, subRuns or runs, or over some other time interval. Three examples of this second sort include the geometry specification, the conditions information[§] and, for simulations, the table of particle properties.

To provide managed access to the second sort of information, *art* supports an idea named *art services* (again, shortened to *services*). Services may also be used to provide certain types of utility functions. Again, a service in *art* is just a C++ class that obeys a set of rules defined by *art*. The rules for services are different than those for modules or data products.

art implements a number of services that it uses for internal functions, a few of which you will encounter in the first couple of Workbook exercises. The *message service*(γ) is used by both *art* and experiment-specific code to limit printout of messages with a low severity level and to route messages to appropriate destinations. It can be configured

[§]The phrase "conditions information" is the currently fashionable name for what was once called "calbration constants"; the name change came about because most calibration information is intrinsically time-dependent, which makes "constants" a poor choice of name.

to provide summary information at the end of the *art* job. The *TFileService*(γ) and the `RandomNumberGenerator` service are not used internally by *art*, but are used by most experiments. Experiments may also create and implement their own services.

After *art* completes its initialization phase and before it constructs any modules (see Section 3.6.2), it

1. reads the configuration to learn what services are requested
2. calls the constructor of each requested service

Once a service has been constructed, any code in any module can ask *art* for a *smart pointer*(γ) to that service and use the features provided by that service. Because services are constructed before modules, they are available for use by modules over the full life cycle of each module.

It is also legal for one service to request information from another service as long as the dependency chain does not have any loops. That is, if Service A uses Service B, then Service B may not use Service A, either directly or indirectly.

For those of you familiar with the C++ Singleton Design Pattern, an *art* service has some differences and some similarities to a Singleton. The most important difference is that the lifetime of a service is managed by *art*, which calls the constructors of all services at a well-defined time in a well-defined order. Contrast this with the behavior of Singletons, for which the order of initialization is undefined by the C++ standard and which is an accident of the implementation details of the loader. *art* also includes services under the umbrella of its powerful run-time configuration system; in the Singleton Design pattern this issue is simply not addressed.

3.6.6 Dynamic Libraries and *art*

When code is executed within the *art* framework, *art*, not the experiment, provides the main executable. The experiment provides its code to the *art* executable in the form of dynamic libraries that *art* loads at run time; these libraries are also called *dynamic load libraries*, *shareable object libraries*, or *plugins*. On Linux, their filenames typically end in `.so`; on OS X, the suffixes `.dylib` and `.so` are both used.

Table 3.1: Compiler flags for the optimization levels defined by **cetbuildtools**; compiler options not related to optimization or debugging are not included in this table.

Name	flags
debug	-O0 -g
prof	-O3 -g -fno-omit-frame-pointer -DNDEBUG
opt	-O3 -DNDEBUG

3.6.7 Build Systems and *art*

To make an experiment's code available to *art*, the source code must be compiled and linked (i.e., *built*) to produce dynamic libraries (Section 3.6.6). The tool that creates the dynamic libraries from the C++ source files is called a *build system*(γ).

Experiments that use *art* are free to choose their own build systems, as long as the system follows the conventions that allow *art* to find the name of the `.so` file given the name of the module class, as discussed in Section 9.4. The Workbook will use a build system named *cetbuildtools*, which is a layer on top of *cmake*[¶].

The **cetbuildtools** system defines three standard compiler optimization levels, called "debug", "profile" and "optimized"; the last two are often abbreviated "prof" and "opt". When code is compiled with the "opt" option, it runs as quickly as possible but is difficult to debug. When code is compiled with the "debug" option, it is much easier to debug but it runs more slowly. When code is compiled with the "prof" option the speed is almost as fast as for an "opt" build and the most useful subset of the debugging information is retained. The "prof" build retains enough debugging information that one may use a profiling tool to identify in which functions the program spends most of its time; hence its name "profile". The "prof" build provides enough information to get a useful traceback from a core dump. Most experiments using *art* use the "prof" build for production and the "debug" build for development.

The compiler options corresponding to the three levels are listed in Table 3.1.

[¶]**cetbuildtools** is also used to build *art* itself.

3.6.8 External Products

As you progress through the Workbook, you will see that the exercises use some software packages that are part of neither *art* nor the toy experiment's code. The Workbook code, *art* and the software for your experiment all rely heavily on some external tools and, in order to be an effective user of *art*-based HEP software, you will need at least some familiarity with them; you may, in fact, need to become expert in some.

These packages and tools are referred to as *external products*(γ) (sometimes called simply *products*).

An initial list of the external products you will need to become familiar with includes:

art the event processing framework

FHiCL the run-time configuration language used by *art*

CETLIB a utility library used by *art*

MF(γ) a message facility that is used by *art* and by (some) experiments that use *art*

ROOT an analysis, data presentation and data storage tool widely used in HEP

CLHEP(γ) a set of utility classes; the name is an acronym for *Class Library for HEP*

boost(γ) a class library with new functionality that is being prototyped for inclusion in future C++ standards

gcc the GNU C++ compiler and run-time libraries; both the core language and the standard library are used by *art* and by your experiment's code.

git(γ) a source code management system that is used for the Workbook and by some experiments; similar in concept to the older CVS and SVN, but with enhanced functionality

cetbuildtools(γ) the build system that is used by the *art* Workbook (and by *art* itself).

UPS(γ) a Fermilab-developed system for accessing software products; it is an acronym for *Unix Product Support*.

UPD(γ) a Fermilab-developed system for distributing software products; it is an acronym for *Unix Product Distribution*.

jobusub_tools(γ) tools for submitting jobs to the Fermigrid batch system and monitoring them.

ifdh_sam(γ) allows *art* to use *SAM*(γ) as an external run-time agent that can deliver remote files to local disk space and can copy output files to tape. SAM is a Fermilab-supplied resource that provides the functions of a file catalog, a replica manager and some functions of a batch-oriented workflow manager

Any particular line of code in a Workbook exercise may use elements from, say, four or five of these packages. Knowing how to parse a line and identify which feature comes from which package is a critical skill. The Workbook will provide a tour of the above packages so that you will recognize elements when they are used and you will learn where to find the necessary documentation.

For the *art* Workbook, external products are made available to your code via a mechanism called UPS, which will be described in Section 7. Many Fermilab experiments also use UPS to manage their external products; this is not required by *art* and you may choose to manage external products whichever way you prefer. UPS is, itself, just another external product. From the point of view of your experiment, *art* is an external product. From the point of view of the Workbook code, both *art* and the code for the toy experiment are external products.

Finally, it is important to recognize an overloaded word, *products*. When a line of documentation simply says *products*, it may be refering either to data products or to external products. If it is not clear from the context which is meant, please let us know (see Section 3.4).

3.6.9 The Event-Data Model and Persistency

Section 3.6.4 introduced the idea of *art* data products. In a small experiment, a fully reconstructed event may contain on the order of ten data products; in a large experiment there may be hundreds.

While each experiment will define its own data product classes, there are many issues that are common to all data products in all experiments:

1. How does my module access data products that are already in the event?
2. How does my module publish a data product so that other modules can see it?
3. How is a data product represented in the memory of a running program?
4. How does an object in one data product refer to an object in another data product?
5. What metadata is there to describe each data product?
   Such metadata might include: which module created it; what was the run-time configuration of that module; what data products were read by that module; what was the code version of the module that created it?
6. How does my module access the metadata associated with a particular data product?

The answers to these questions form what is called the *Event-Data Model*(γ) (EDM) that is supported by the framework.

A question that is closely related to the EDM is: what technologies are supported to write data products from memory to a disk file and to read them from the disk file back into memory in a separate *art* job? A framework may support several such technologies. *art* currently supports only one disk file format, a ROOT-based format, but the *art* EDM has been designed so that it will be straightforward to support other disk file formats as it becomes useful to do so.

A few other related terms that you will encounter include:

1. *transient representation*: the in-memory representation of a data product
2. *persistent representation*: the on-disk representation of a data product
3. *persistency*: the technology to convert data products back and forth between their persistent and transient representations

3.6.10 Event-Data Files

When you read data from an experiment and write the data to a disk file, that disk file is usually called a *data file*.

When you simulate an experiment and write a disk file that holds the information produced by the simulation, what should you call the file? The Particle Data Group has recommended that this not be called a "data file" or a "simulated data file;" they prefer that the word "data" be strictly reserved for information that comes from an actual experiment. They recommend that we refer to these files as "files of simulated events" or "files of Monte Carlo events" [‖]. Note the use of "events," not "data."

This leaves us with a need for a collective noun to describe both data files and files of simulated events. The name in current use is *event-data files*(γ); yes this does contain the word "data" but the hyphenated word, "event-data", is unambiguous and this has become the standard name.

3.6.11 Files on Tape

Many experiments do not have access to enough disk space to hold all of their event-data files, ROOT files and log files. The solution is to copy a subset of the disk files to tape and to read them back from tape as necessary.

At any given time, a snapshot of an experiment's files will show some on tape only, some on tape with copies on disk, and some on disk only. For any given file, there may be multiple copies on disk and those copies may be distributed across many *sites*(γ), some at Fermilab and others at collaborating laboratories or universities.

Conceptually, two pieces of software are used to keep track of which files are where, a *File Catalog* and a *Replica Manager*. One software package that fills both of these roles is called SAM, which is an acronym for "Sequential data Access via Metadata." SAM also provides some tools for Workflow management. SAM is in wide use at Fermilab and you can learn more about SAM at:
https://cdcvs.fnal.gov/redmine/projects/sam-main/wiki.

[‖] In HEP almost all simulations codes use *Monte Carlo*(γ) methods; therefore simulated events are often referred to as *Monte Carlo events* and the simulation process is referred to as *running the Monte Carlo*.

3.7 The Toy Experiment

The Workbook exercises are based around a made-up (*toy*) experiment. The code for the toy experiment is deployed as a UPS product named *toyExperiment*. The rest of this section will describe the physics content of toyExperiment; the discussion of the code in the **toyExperiment** UPS product will unfold in the Workbook, in parallel to the exposition of *art*.

The software for the toy experiment is designed around a toy detector, which is shown in Figure 3.3. The *toyExperiment* code contains many C++ classes: some modules, some data products, some services and some plain old C++ classes. About half of the modules are producers that individually perform either one step of the simulation process or one step of the reconstruction/analysis process. The other modules are analyzers that make histograms and ntuples of the information produced by the producers. There are also event display modules.

3.7.1 Toy Detector Description

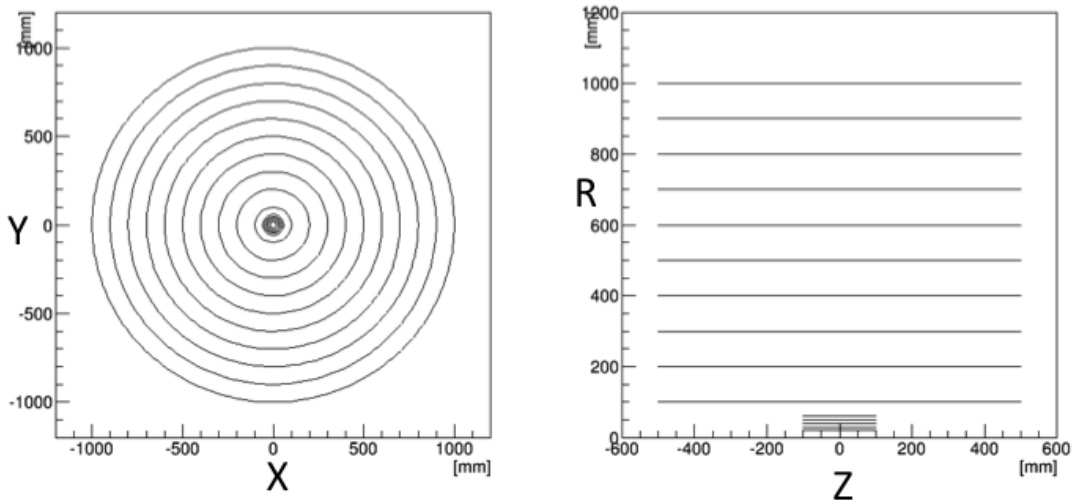


Figure 3.3: The geometry of the toy detector; the figures are described in the text. A uniform magnetic field of strength 1.5 T is oriented in the $+z$ direction.

The toy detector is a central detector made up of 15 concentric shells, with their axes

Table 3.2: Units used in the Workbook

Quantity	Unit
Length	mm
Energy	MeV
Time	ns
Plane Angle	radian
Solid Angle	steradian
Electric Charge	Charge of the proton = +1
Magnetic Field	Tesla

centered on the z axis; the left-hand part of Figure 3.3 shows an xy view of these shells and the right shows the radius vs z view. The inner five shells are closely spaced radially and are short in z; the ten outer shells are more widely spaced radially and are longer in z. The detector sits in a uniform magnetic field of 1.5 T oriented in the $+z$ direction. The origin of the coordinate system is at the center of the detector. The detector is placed in a vacuum.

Each shell is a detector that measures (φ, z), where φ is the azimuthal angle of a line from the origin to the measurement point. Each measurement has perfectly gaussian measurement errors and the detector always has perfect separation of hits that are near to each other. The geometry of each shell, its efficiency and resolution are all configurable at run-time.

All of the code in the toyExperiment product works in the set of units described in Table 3.2. Because the code in the Workbook is built on toyExperiment, it uses the same units. *art* itself is not unit-aware and places no constraints on which units your experiment may use.

The first six units listed in Table 3.2 are the base units defined by the CLHEP SystemOfUnits package. These are also the units used by Geant4.

3.7.2 Workflow for Running the Toy Experiment Code

The workflow of the toy experiment code includes five steps: three simulation steps, a reconstruction step and an analysis step:

1. event generation
2. detector simulation
3. hit-making
4. track reconstruction
5. analysis of the mass resolution

For each event, the event generator creates some signal particles and some background particles. The first signal particle is generated with the following properties:

- Its mass is the rest mass of the ϕ meson; the event generator does not simulate a natural width for this particle.
- It is produced at the origin.
- It has a momentum that is chosen randomly from a distribution that is uniform between 0 and 2000 MeV/c.
- Its direction is chosen randomly on the unit sphere.

The event generator then decays this particle to K^+K^-; the center-of-mass decay angles are chosen randomly on the unit sphere.

The background particles are generated by the following algorithm:

- Background particles are generated in pairs, one π^+ and one π^-.
- The number of pairs in each event is a random variate chosen from a Poisson distribution with a mean of 0.75.
- Each of the pions is generated as follows:
  - It is produced at the origin.
  - It has a momentum that is chosen randomly from a distribution that is uniform between 0 and 800 MeV/c.

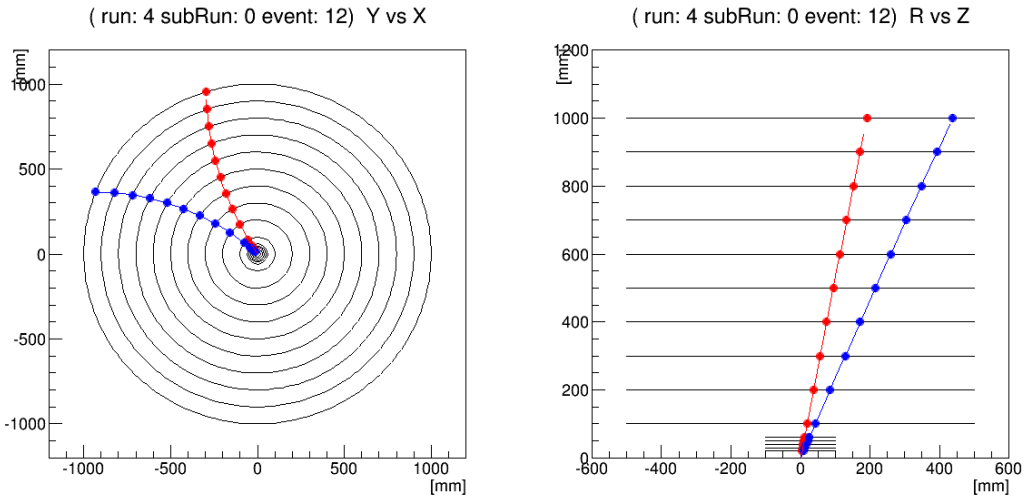


Figure 3.4: Event display of a simulated event in the toy detector.

– Its direction is chosen randomly on the unit sphere.

The above algorithm generates events with a total charge of zero but there is no concept of momentum or energy balance. About 47% of these events will not have any background tracks.

In the detector simulation step, particles neither scatter nor lose energy when they pass through the detector cylinders; nor do they decay. Therefore, the charged particles follow a perfectly helical trajectory. The simulation follows each charged particle until it either exits the detector or until it completes the outward-going arc of the helix. When the simulated trajectory crosses one of the detector shells, the simulation records the true point of intersection. All intersections are recorded; at this stage in the simulation, there is no notion of inefficiency or resolution. The simulation does not follow the trajectory of the ϕ meson because it was decayed in the generator.

Figure 3.4 shows an event display of a simulated event that has no background tracks. In this event the ϕ meson was travelling close to 90° to the z axis and it decayed nearly symmetrically; both tracks intersect all 15 detector cylinders. The left-hand figure shows an xy view of the event; the solid lines show the trajectory of the kaons, red for K^+ and blue for K^-; the solid dots mark the intersections of the trajectories with the detector shells. The right-hand figure shows the same event but in an rz view.

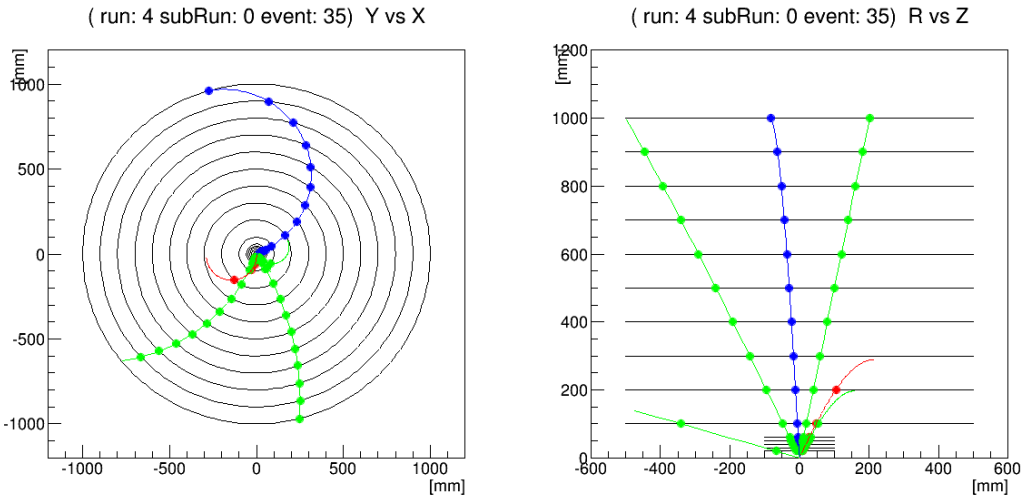


Figure 3.5: Event display of another simulated event in the toy detector; a K^- (blue) is produced with a very shallow trajectory and it does not intersect any detector shells while the K^+ (red) makes five hits in the inner detector and seven in the outer detector

Figure 3.5 shows an event display of another simulated event, one that has four background tracks, all drawn in green. In the xy view it is difficult to see the two π^- tracks, which have very low transverse momentum, but they are clear in the rz view. Look at the K^+ track, draw in red; its trajectory just stops in the middle of the detector. Why does this happen? In order to keep the exercises focused on *art* details, not geometric corner cases, the simulation stops a particle when it completes the outward-going arc of the helix and starts to curl back towards the z axis; it does this even if the the particle is still inside the detector.

The third step in the simulation chain (hit-making) is to inspect the intersections produced by the detector simulation and turn them into data-like hits. In this step, a simple model of inefficiency is applied and some intersections will not produce hits. Each hit represents a 2D measurement (φ, z); each component is smeared with a gaussian distribution.

The three simulation steps use tools provided by *art* to record the *truth information*(γ) about each hit. Therefore it is possible to navigate from any hit back to the intersection from which it is derived, and from there back to the particle that made the intersection.

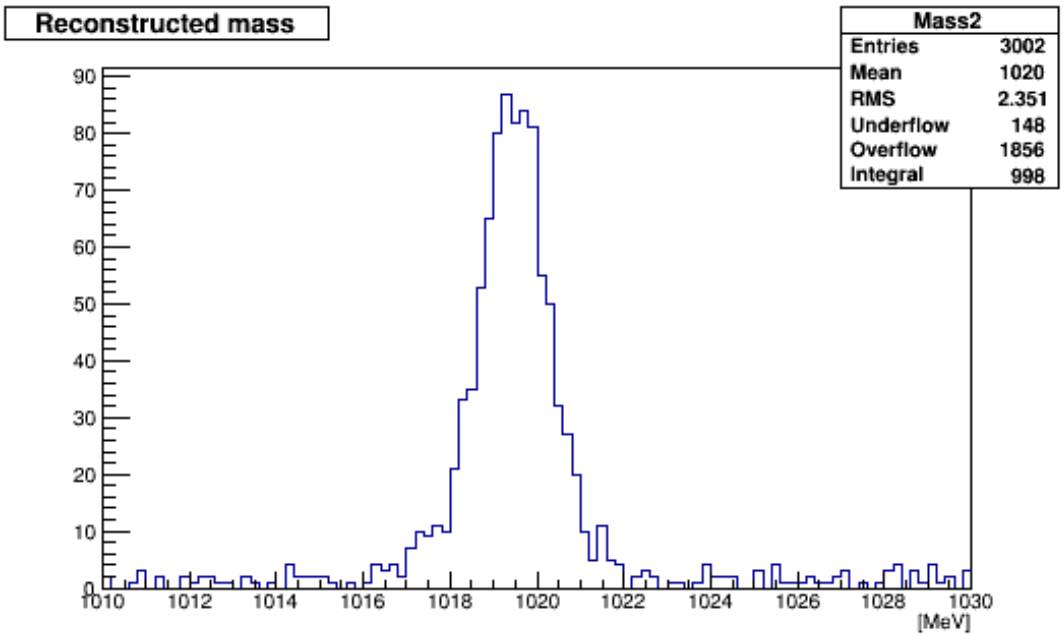


Figure 3.6: The invariant mass of all reconstructed pairs of oppositely charged tracks; for this all reconstructed tracks are assumed to be kaons.

The fourth step is the reconstruction step. The toyExperiment does not yet have properly working reconstruction code; instead it mocks up credible looking results. The output of this code is a data product that represents a fitted helix; it contains the fitted track parameters of the helix, their covariance matrix and collection of smart pointers that point to the hits that are on the reconstructed track. When we write proper tracking finding and track fitting code for the toyExperiment, the classes that describe the fitted helix will not change. Because the main point of the Workbook exercises is to illustrate the bookkeeping features in *art*, this is good enough for the task at hand. The mocked-up reconstruction code will only create a fitted helix object if the number of hits on a track is greater than some minimum value. Therefore there may be some events in which the output data product is be empty.

The fifth step in the workflow does a simulated analysis using the fitted helices from the reconstruction step. It forms all distinct pairs of tracks and requires that they be oppositely charged. It then computes the invariant mass of the pair, under the assumption that both

fitted helices are kaons.** This module is an analyzer module and does not make any output data product. But it does make some histograms, one of which is a histogram of the reconstructed invariant mass of all pairs of oppositely charged tracks; this histogram is shown in Figure 3.6. When you run the Workbook exercises, you will make this plot and can compare it to Figure 3.6. In the figure you can see a clear peak that is created when the two reconstructed tracks are the two true daughters of the generated φ meson. You can also see an almost flat contribution that occurs when at least one of the reconstructed tracks comes from one of the generated background particles.

3.8 Rules, Best Practices, Conventions and Style

In many places, the Workbook will recommend that you write fragments of code in a particular way. The reason for any particular recommendation may be one of the following:

- It is a hard rule enforced by the C++ language or by one of the external products.
- It is a recommended best practice that might not save you time or effort now but will in the long run.
- It is a convention that is widely adopted; C++ is a rich enough language that it will let you do some things in many different ways. Code is much easier to understand and debug if an experiment chooses to always write code fragments with similar intent using a common set of conventions.
- It is simply a question of style.

It is important to be able to distinguish between rules, best practices, conventions and styles; you must follow the rules; it wise to use best practices and established conventions; but style suggestions are just that, suggestions. This documentation will distinguish among these options when discussing the recommendations that it makes.

If you follow the recommendations for best practices and common conventions, it will be easier to verify that your code is correct and your code will be easier to understand, develop and maintain.

**The toy experiment does not have any particle identification system so analysis code cannot know if a reconstructed track is a pion or a kaon. A planned enhancement of the toy experiment is to add a particle identification device.

4 Unix Prerequisites

4.1 Introduction

You will work through the Workbook exercises on a computer that is running some version of the Unix operating system. This chapter describes where to find information about Unix and gives a list of Unix commands that you should understand before starting the Workbook exercises. This chapter also describes a few ideas that you will need immediately but which are usually not covered in the early chapters of standard Unix references.

If you are already familiar with Unix and the *bash*(γ) shell, you can safely skip this chapter.

4.2 Commands

In the Workbook exercises, most of the commands you will enter at the Unix prompt will be standard Unix commands, but some will be defined by the software tools that are used to support the Workbook. The non-standard commands will be explained as they are encountered. To understand the standard Unix commands, any standard Linux or Unix reference will do. Section 4.10 provides links to Unix references.

Most Unix commands are documented via the *man page* system (short for "manual"). To get help on a particular command, type the following at the command prompt, replacing <command-name> with the actual name of the command:

```
man <command-name>
```

In Unix, everything is case sensitive; so the command man must be typed in lower case. You can also try the following; it works on some commands and not others:

`<command-name> --help`

or

`<command-name> -?`

Before starting the Workbook, make sure that you understand the basic usage of the following Unix commands:

cat, cd, cp, echo, export, gzip, head, less, ln -s, ls,

mkdir, more, mv, printenv, pwd, rm, rmdir, tail, tar

You also need to be familiar with the following Unix concepts:

- filename vs pathname
- absolute path vs relative path
- directories and subdirectories (equivalent to folders in the Windows and Mac worlds)
- current working directory
- home directory (aka login directory)
- `../` notation for viewing the directory above your current working directory
- environment variables (discussed briefly in Section 4.5)
- *paths*(γ) (in multiple senses; see Section 4.6)
- file protections (read-write-execute, owner-group-other)
- symbolic links
- stdin, stdout and stderr
- redirecting stdin, stdout and stderr
- putting a command *in the background* via the & character
- pipes

4.3 Shells

When you type a command at the prompt, a Unix agent called a *Unix shell*, or simply a *shell*, reads your command and figures out what to do. Some commands are executed internally by the shell but other commands are dispatched to an appropriate program or script. A shell lives between you and the underlying operating system; most versions of Unix support several shells. The *art* Workbook code expects to be run in the *bash shell*. You can see which shell you're running by entering:

```
echo $SHELL
```

For those of you with accounts on a Fermilab machine, your login shell was initially set to the *bash* shell*.

If you are working on a non-Fermilab machine and *bash* is not your default shell, consult a local expert to learn how to change your login shell to bash.

4.4 Scripts: Part 1

In order to automate repeated operations, you may write multiple Unix commands into a file and tell *bash* to run all of the commands in the file as if you had typed them sequentially. Such a file is an example of a *shell script* or a *bash script*. The *bash* scripting language is a powerful language that supports looping, conditional execution, tests to learn about properties of files and many other features.

Throughout the Workbook exercises you will run many scripts. You should understand the big picture of what they do, but you don't need to understand the details of how they work.

If you would like to learn more about *bash*, some references are listed in Section 4.10.

* If you have had a Fermilab account for many years, your default shell might be something else. If your default shell is not bash, open a Service Desk ticket to request that your default shell be changed to bash.

4.5 Unix Environments

4.5.1 Building up the Environment

Very generally, a Unix *environment* is a set of information that is made available to programs so that they can find everything they need in order to run properly. The Unix operating system itself defines a generic environment, but often this is insufficient for everyday use. However, an environment sufficient to run a particular set of applications doesn't just pop out of the ether, it must be *established* or *set up*, either manually or via a script. Typically, on institutional machines at least, system administrators provide a set of login scripts that run automatically and enhance the generic Unix environment. This gives users access to a variety of system resources, including, for example:

- disk space to which you have read access
- disk space to which you have write access
- commands, scripts and programs that you are authorized to run
- proxies and tickets that authorize you to use resources available over the network
- the actual network resources that you are authorized to use, e.g., tape drives and DVD drives

This constitutes a basic *working environment* or *computing environment*. Environment information is largely conveyed by means of *environment variables* that point to various program executable locations, data files, and so on. A simple example of an environment variable is HOME, the variable whose value is the absolute path to your home directory.

Particular programs (e.g., *art*) usually require extra information, e.g., paths to the program's executable(s) and to its dependent programs, paths indicating where it can find input files and where to direct its output, and so on. In addition to environment variables, the *art*-enabled computing environment includes some aliases and *bash* functions that have been defined; these are discussed in Section 4.8.

In turn, the Workbook code, which must work for all experiments and at Fermilab as well as at collaborating institutions, requires yet more environment configuration – a *site-specific* configuration.

Given the different experiments using *art* and the variety of laboratories and universities at which the users work, a *site*(γ) in *art* is a unique combination of *experiment* and *institution*. It is used to refer to a set of computing resources configured for use by a particular experiment at a particular institution. Setting up your site-specific environment will be discussed in Section 4.7.

When you finish the Workbook and start to run real code, you will set up your experiment-specific environment on top of the more generic *art*-enabled environment, in place of the Workbook's. To switch between these two environments, you will log out and log back in, then run the script appropriate for the environment you want. Because of potential naming "collisions," it is not guaranteed that these two environments can be overlain and always work properly.

This concept of the environment hierarchy is illustrated in Figure 4.1.

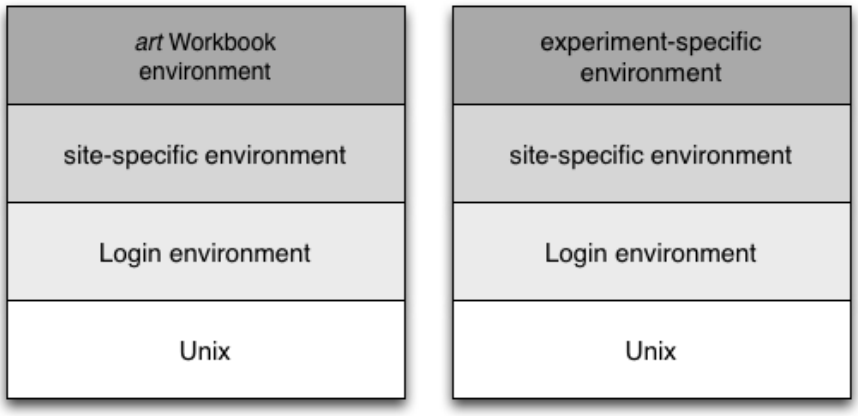


Figure 4.1: Components of the *art* Workbook (left) and experiment-specific (right) computing environments, shown in the order in which they are constructed, starting with the Unix environment

4.5.2 Examining and Using Environment Variables

One way to see the value of an environment variable is to use the printenv command:

```
printenv HOME
```

At any point in an interactive command or in a shell script, you can tell the shell that

you want the value of the environment variable by prefixing its name with the $ character:

```
echo $HOME
```

Here, echo is a standard Unix command that copies its arguments to its output, in this case the screen.

By convention, environment variables are virtually always written in all capital letters[†].

There may be times when the Workbook instructions tell you to set an environment variable to some value. To do so, type the following at the command prompt:

```
export <ENVNAME>=<value>
```

If you read *bash* scripts written by others, you may see the following variant, which accomplishes the same thing:

```
<ENVNAME>=<value>
export <ENVNAME>
```

4.6 Paths and $PATH

Path (or *PATH*) is an overloaded word in computing. Here are the ways in which it is used:

Lowercase path can refer to the location of a file or a directory; a path may be absolute or relative, e.g.
`/absolute/path/to/mydir/myfile` or
`relative/path/to/mydir/myfile` or
`../another/relative/path/to/mydir/myfile`

PATH refers to the standard Unix environment variable set by your login scripts and updated by other scripts that extend your environment; it is a colon-separated list of directory names, e.g.,
`/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin`.

[†] Another type of variable, *shell variables*, are local to the current shell and are not copied to sub shells. By convention, these are written in lower or mixed case. These conventions provide a clue to the programmer as to whether changing a variable's value might have consequences outside the current shell.

It contains the list of directories that the shell searches to find programs/files required by Unix shell commands (i.e., PATH is used by the shell to "resolve" commands).

"path" generically, refers to any environment variable whose value is a colon-separated list of directory names e.g.,
`/abs/path/a:/abs/path/b:rel/path/c`

In addition, *art* defines a fourth idea, also called a path, that is unrelated to any of the above; it will be described as you encounter it in the Workbook, e.g., Section 9.8.8.

All of these path concepts are important to users of *art*. In addition to PATH itself, there are three PATH-like environment variables (colon-separated list of directory names) that are particularly important:

`LD_LIBRARY_PATH` (Linux only) used by *art* to resolve dynamic libraries

`DYLD_LIBRARY_PATH` (OS X only) used by *art* to resolve dynamic libraries

`PRODUCTS` used by UPS to resolve external products

`FHICL_FILE_PATH` use by FHiCL to resolve `#include` directives.

When you source the scripts that setup your environment for *art*, these will be defined and additional colon-separated elements will be added to your PATH. To look at the value of PATH (or the others), enter:

```
printenv PATH
```

To make the output easier to read by replacing all of the colons with newline characters, enter:

```
printenv PATH | tr : \\n
```

In the above line, the vertical bar is referred to as a *pipe* and `tr` is a standard Unix command. A pipe takes the output of the command to its left and makes that the input of the command to its right. The `tr` command replaces patterns of characters with other patterns of characters; in this case it replaces every occurrence of the colon character with the newline character. To learn why a double back slash is needed, read bash documentation to learn about escaping special characters.

4.7 Scripts: Part 2

There are two ways to run a bash script (actually three, but two of them are the same). Suppose that you are given a bash script named `file.sh`. You can run any of these commands:

```
file.sh
```

```
source file.sh
```

```
. file.sh
```

The first version, `file.sh`, starts a new bash shell, called a subshell, and it executes the commands from `file.sh` in that subshell; upon completion of the script, control returns to the parent shell. At the startup of a subshell, the environment of that subshell is initialized to be a copy of the environment of its parent shell. If `file.sh` modifies its environment, then it will modify only the environment of the subshell, leaving the environment of the parent shell unchanged. This version is called *executing* the script.

The second and third versions are equivalent. They do not start a subshell; they execute the commands from `file.sh` in your current shell. If `file.sh` modifies any environment variables, then those modifications remain in effect when the script completes and control returns to the command prompt. This is called *sourcing* the script.

Some shell scripts are designed so that they must be sourced and others are designed so that they must be executed. Many shell scripts will work either way.

If the purpose of a shell script is to modify your working environment then it must be sourced, not executed. As you work through the Workbook exercises, pay careful attention to which scripts it tells you to source and which to execute. In particular, the scripts to setup your environment (the first scripts you will run) are bash scripts that must be sourced because their purpose is to configure your environment so that it is ready to run the Workbook exercises.

Some people adopt the convention that all bash scripts end in `.sh`; others adopt the convention that only scripts designed to be sourced end in `.sh` while scripts that must be executed have no file-type ending (no ".something" at the end). Neither convention is uniformly applied either in the Workbook or in HEP in general.

If you would like to learn more about bash, some references are listed in Section 4.10.

4.8 bash Functions and Aliases

The bash shell also has the notion of a *bash function*. Typically bash functions are defined by sourcing a bash script; once defined, they become part of your environment and they can be invoked as if they were regular commands. The setup <product> "command" that you will sometimes need to issue, described in Chapter 7, is an example. A bash function is similar to a bash script in that it is just a collection of bash commands that are accessible via a name; the difference is that bash holds the definition of a function as part of the environment while it must open a file every time that a bash script is invoked.

You can see the names of all defined bash functions using:

```
declare -F
```

The bash shell also supports the idea of *aliases*; this allows you to define a new command in terms of other commands. You can see the definition of all aliases using:

```
alias
```

You can read more about bash shell functions and aliases in any standard bash reference.

When you type a command at the command prompt, bash will resolve the command using the following order:

1. Is the command a known alias?
2. Is the command a bash keyword, such as if or declare?
3. Is the command a shell function?
4. Is the command a shell built-in command?
5. Is the command found in $PATH?

To learn how bash will resolve a particular command, enter:

```
type <command-name>
```

4.9 Login Scripts

When you first login to a computer running the Unix operating system, the system will look for specially named files in your home directory that are scripts to set up your working environment; if it finds these files it will source them before you first get a shell prompt. As mentioned in Section 4.5, these scripts modify your PATH and define bash functions, aliases and environment variables. All of these become part of your environment.

When your account on a Fermilab computer was first created, you were given standard versions of the files `.profile` and `.bashrc`; these files are used by bash[‡]. You can read about login scripts in any standard bash reference. You may add to these files but you should not remove anything that is present.

If you are working on a non-Fermilab computer, inspect the login scripts to understand what they do.

It can be useful to inspect the login scripts of your colleagues to find useful customizations.

If you read generic Unix documentation, you will see that there are other login scripts with names like, `.login`, `.cshrc` and `.tcshrc`. These are used by the csh family of shells and are not relevant for the Workbook exercises, which require the bash shell.

4.10 Suggested Unix and bash References

The following cheat sheet provides some of the basics:

- http://mu2e.fnal.gov/atwork/computing/UnixHints.shtml

A more comprehensive summary is available from:

- http://www.tldp.org/LDP/.../GNU-Linux-Tools-Summary.html

Information about writing bash scripts and using bash interactive features can be found in:

- BASH Programming - Introduction HOW-TO
  http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

[‡]These files are used by the sh family of shells, which includes bash.

- Bash Guide for Beginners
  http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html
- Advanced Bash Scripting Guide
  http://www.tldp.org/LDP/abs/html/abs-guide.html

The first of these is a compact introduction and the second is more comprehensive.

The above guides were all found at the Linux Documentation Project: Workbook:

- http://www.tldp.org/guides.html

5 Site-Specific Setup Procedure

Section 4.5 discussed the notion of a working environment on a computer. This chapter answers the question: How do I make sure that my environment variables are set correctly to run the Workbook exercises or my experiment's code using *art*?

Very simply, on every computer that hosts the Workbook, a procedure must be established that every user is expected to follow once per login session. In most cases (NOνA being a notable exception), the procedure involves only sourcing a shell script (recall the discussion in Section 4.7). In this documentation, we refer to this procedure as the "site-specific setup procedure." It is the responsibility of the people who maintain the Workbook software for each *site*(γ) to ensure that this procedure does the right thing on all the site's machines.

As a user of the Workbook, you will need to know what the procedure is and you must remember to follow it each time that you log in.

For all of the Intensity Frontier experiments at Fermilab, the site-specific setup procedure defines all of the environment variables that are necessary to create the working environment for either the Workbook exercises or for the experiment's own code.

Table 5.1 lists the site-specific setup procedure for each experiment. You will follow the procedure when you get to Section 9.6.

For NOvA, until further notice, you need to add the bash function definition from Listing 5.1 to your login scripts. To run the site-specifc setup procedure, type the following at the command line:

```
setup_art_workbook
```

Table 5.1: Site-specific setup procedures for Experiments that run *art*; for the equivalent procedure at a non-Fermi site, consult an expert from that site. The MicroBoone experiment maintains two copies of its setup scripts, one on `/grid/fermiapp` and the other in CVMFS space; if you can see both from your computer, they are equivalent.

Experiment	Site-Specific Setup Procedure
ArgoNeut	See the instructions for MicroBoone
Darkside	`source /ds50/app/ds50/ds50.sh`
LArIAT	Will be available in the next release of the workbook
LBNE	`source /grid/fermiapp/lbne/software/setup_lbne.sh`

MicroBoone	`source /grid/fermiapp/products/uboone/setup_uboone.sh`
	or
	`source /cvmfs/oasis.opensciencegrid.org/microboone/products/setup_uboone.sh`

Muon g-2	`source /gm2/app/software/prod/g-2/setup`
Mu2e	`setup mu2e`
NOνA	See the text

Listing 5.1: NOvA `setup_art_workbook` script

```
1 function setup_art_workbook
2 {
3     echo "Setting up art workbook"
4     source /grid/fermiapp/nova/novaart/novasvn/srt/srt.sh
5     export EXTERNALS=/nusoft/app/externals
6     export ART_WORKBOOK_OUTPUT_BASE=/nova/data/users
7     export ART_WORKBOOK_WORKING_BASE=/nova/app/users
8     export ART_WORKBOOK_QUAL="nu:e5"
9 }
```

6 Get your C++ up to Speed

6.1 Introduction

There are two goals for this chapter. The first is to illustrate the features of C++ that will be important for users of the Workbook, especially those features that will be used in the first few Workbook exercises. It does not attempt to cover C++ comprehensively and it delegates as much as possible to the standard documentation.

The second goal is to explain the process of turning source code files into an *executable program*. The two steps in this process are *compiling* and *linking*. In informal writing, the word *build* is sometimes used to mean just compiling or just linking, but usually it refers to the two together.

A typical program consists of many source code files, each of which contains a human-readable description of one component of the program. In the Workbook, you will see source code files written in the C++ computer language; these files have names that end in `.cc`. In C++, there is a second sort of source code file, called a *header file* that ends in `.h`; in most, but not all, cases for every file ending in `.cc` there is another file with the same name but ending in `.h`. Header files can be thought of as the "parts list" for the corresponding `.cc` file; you will see how these are used in Section 6.4.

In the compilation step each `.cc` file is translated into *machine code*, also called *binary code* or *object code*, which is a set of instructions, in the computer's native language, to do the tasks described by the source code. The output of the compilation step is called an *object file*; in the examples you will see in the Workbook, object files always end in `.o`. But an object file, by itself, is not an *executable program*. It is not executable because each `.o` file was created in isolation and does not know about the other `.o` files.

It is often convenient to collect related groups of `.o` files and put them into *libraries*. There are two kinds of library files, static libraries, whose names end in `.a` and dynamic libraries whose names — on systems suported by *art*— end in `.so` or `.dylib`. Putting many `.o` files into a single library allows you to use them as a single coherent entity. We will defer further discussion of libraries until more background information has been provided.

The job of the *linking* step is to read the information found in the various libraries and `.o` files and form them into an *executable program*. When you run the linker, you tell it the name of the file into which it will write the executable program. It is a common, but not universal, practice that the filename of an executable program has no extension (i.e., no `.something` at the end).

After the linker has finished, you can run your executable program typing the filename of the program at the bash command prompt.

A typical program links both to libraries that were built from the program's source code and to libraries from other sources. Some of these other libraries might have been developed by the same programmer as general purpose tools to be used by his or her future programs; other libraries are provided by third parties, such as *art* or your experiment. Many C++ language features are made available to your program by telling the linker to use libraries provided by the C++ compiler vendor. Other libraries are provided by the operating system.

Now that you know about libraries, we can give a second reason why an object file, by itself, is not an executable program: until it is linked, it does not have access to the functions provided by any of the external libraries. Even the simplest program will need to be linked against some of the libraries supplied by the compiler vendor and by the operating system.

The names of all of the libraries and object files that you give to the linker is called the *link list*.

This chapter is designed around a handful of exercises, each of which you will first build and run, then "pick apart" to understand how the results were obtained.

6.2 Establishing the Environment

6.2.1 Initial Setup

To start these exercises for the first time, do the following:

1. Log into the node that you will use for Workbook exercises.
2. Follow the site-specific setup procedure from Table 5.1.
3. Create an empty working directory and cd to it.
4. Run these commands (the first one is shown on two lines) to copy a gzipped tar file from the web, unpack it, and get a directory listing:

```
wget https://web.fnal.gov/project/\
  ArtDoc/Shared%20Documents/C++UpToSpeed.tar.gz

tar xzf C++UpToSpeed.tar.gz

rm C++UpToSpeed.tar.gz

ls
BasicSyntax Build Classes Libraries setup.sh
```

5. To select the correct compiler version and define a few environment variables that will be used later in these exercises, run:

```
source setup.sh
```

After these steps, you are ready to begin the exercise in Section 6.3.

6.2.2 Subsequent Logins

If you log out and log back in again, reestablish your environment by following these steps:

1. Log into the node that you will normally use.
2. Follow the site-specific setup procedure.
3. cd to the working directory you created in Section 6.2.1.
4. Run the command: `source setup.sh`
5. cd to the directory that contains the exercise you want to work on.

6.3 C++ Exercise 1: The Basics

6.3.1 Concepts to Understand

This section provides a program that illustrates the parts of C++ that are assumed knowledge for the Workbook material. If you do not understand some of the code in this example program, consult any standard C++ reference; several are listed in Section 6.7.

Once you have understood this example program, you should understand the following C++ concepts:

- how comments are indicated
- what is a *main program*
- how to write a main program
- how to compile, link and run the main program
- how to distinguish between source, object and executable files
- how to print to standard output, `std::cout`
- how to declare and define *variables*(γ) of the some of the frequently used built-in types: `int, float, double, bool`
- the {} initializer syntax
- assignment to variables
- arrays
- several different forms of looping

- comparisons: `==, !=, <, >, >=, <=`
- `if-then-else, if-then-else if-else`
- pointers
- references
- `std::string` (a type from the C++ Standard Library (*std*(γ))
- the class template from the standard library, `std::vector<T>`*

The above list explicitly does not include classes, objects and inheritance, which will be discussed in Sections 6.6 and a future section on inheritance.

6.3.2 How to Compile, Link and Run

In this section you will learn how to compile, link and run the small C++ program that illustrates the features of C++ that are considered prerequisites. The main discussion of the details of compiling and linking will be deferred until Section 6.4.

We don't offer a lot of details up front; more will follow in Sections 6.3.5 and 6.3.4. The idea here is to get used to the steps and see what results you get.

To compile, link and run the sample C++ program, called `t1`:

1. If not yet done, log in and establish the working environment (Section 6.2).
2. List the starting set of files:

```
cd BasicSyntax/v1/

ls
build t1.cc t1_example.log
```

*You need to know how to use `std::vector<T>` but you do not need to understand how it works or how to write your own templates.

The file `t1.cc` contains the source code of the *main program*, which is a function called `main() { ...}`. The file `build` is a script that will compile and link the code. The file `t1_example.log` is an example of the output expected when you run t1.

3. Compile and link the code (one step); then look at a directory listing:

```
build
t1.cc: In function 'int main()':
t1.cc:43:26: warning: 'k' may be
used uninitialized in this function
[-Wuninitialized]

ls
build t1 t1.cc t1_example.log
```

The script named `build` compiles and links the code, and produces the executable file `t1`. The warning message, issued by the compiler, also comes during this step.

4. Run the executable file sending output to a log file:

```
./t1 > t1.log
```

6.3.3 Suggested Homework

1. Compare your output with the standard example:

```
diff t1.log t1_example.log
```

There will almost certainly be a handful of differences.

2. Look at the file `t1.cc` and understand what it does, in particular the relationship

between the lines in the program and the lines in the output.

If you don't understand something, consult a standard C++ reference; see Section 6.7. A few of your questions might also be answered in Section 6.3.4.

6.3.4 Discussion

Why do we expect several of the lines of the output to be different from those in `t1_example.log`? There are two classes of answers: (1) an uninitialized variable and (2) variation in variable addresses from run to run.

In `t1.cc`, the line `int k;` declares that `k` is a variable whose type is `int` but it does not initialize the variable. Therefore the value of the variable `k` is whatever value happened to be sitting in the memory location that the program assigned to `k`. Each time that the program runs, the operating system will put the program into whatever region of memory makes sense to the operating system; therefore the address of any variable, and thus the value returned, may change unpredictably from run to run.

This line is also the source of the warning message produced by the `build` script. This line was included to make it clear what we mean by *initialized* variables and *uninitialized* variables. Uninitialized variables are frequent sources of errors in code and therefore you should *always* initialize your variables. In order to help you establish this good coding habit, the remaining exercises in this series and in the Workbook include the compiler option `-Werror`. This tells the compiler to promote warning messages to error level and to stop compilation without producing an output file.

The second line that may differ from one run to the next is:

```
float *pa=&a;
```

This line declares a variable `pa`, which is of type *pointer*(γ) to float, and it initializes it to be the memory address of the variable `a` (`a` must be of type float).

This line could have been written with the asterisk next to `float`:

```
float* pa=&a;
```

Some people may find this easier to understand conceptually because every term in the expression is of the same type, i.e. a memory address of a float.

Since the address may change from run to run, so may the printout that starts `pa =`.

For similar reasons, the lines in the printout that start `&a =` and `&ra =` may also change from run to run.

6.3.5 How was this Exercise Built?

Just to see how the exercise was built, look at the script `BasicSyntax/v1/build` that you ran to compile and link `t1.cc`; the following command was issued:

```
c++ -Wall -Wextra -pedantic -std=c++11 -o t1 t1.cc
```

This turned the source file `t1.cc` into an executable program, named `t1` (the argument to the -o (for "output") option). We will discuss compiling and linking in Section 6.4.

6.4 C++ Exercise 2: About Compiling and Linking

6.4.1 What You Will Learn

In the previous exercise, the entire program was found in a single file and the build script performed compiling and linking in a single step. For all but the smallest programs, this is not practical. It would mean, for example, that you would need to recompile and relink everything when you made even the smallest change anywhere in the code; generally this would take much too long. To address this, some computer languages, including C++, allow you to break up a large program into many smaller files and rebuild only a small subset of files when you make changes in one.

There are two exercises in this section. In the first one the source code consists of three files. This example has enough richness to discuss the details of what happens during compiling and linking, without being overwhelming. The second exercise introduces the ideas of libraries and external packages.

6.4.2 The Source Code for this Exercise

The source code for this exercise is found in `Build/v1`, relative to your working directory. The relevant files are `function.cc`, `function.h` and `t1.cc`.

The file `t1.cc` contains the source code for the function `main() { ...}` for this exercise. Every C++ program must have one and only one function named `main`, which is where the program actually starts execution. Note that the term *main program* sometimes refers to this function, but other times refers to the `.cc` file that contains it. In either case, *main program* refers to this function, either directly or indirectly. For more information, consult any standard C++ reference. The file `function.h` is a header file that declares a function named `function`. The file `function.cc` is another source code file; it provides the definition of that function.

Look at `t1.cc`: it both declares and defines the program's function `main() { ... }` that takes no arguments. A function with this *signature*(γ) has special meaning to the complier and the linker: they recognize it as a C++ *main program*. There are other signatures that the compiler and linker will recognize as a C++ main program; consult the standard C++ documentation.

To be recognized as a main program, there is one more requirement: `main() { ... }` must be declared in the global namespace.

The body of the main program (between the braces), declares and defines a variable `a` and initializes it to the value of 3; it prints out the value of `a`. Then it calls a function that takes `a` as an argument and prints out the value returned by that function.

You, as the programmer using that function, need to know what the function does but the C++ compiler doesn't. It only needs to know the name, argument list and return type of the function — information that is provided in the header file, `function.h`. This file contains the line

```
1  float function( float );
```

This line is called the *declaration*(γ) of the function. It says (1) that the identifier `function` is the name of a function that (2) takes an argument of type `float` (the "float" inside the parentheses) and (3) returns a value of type `float` (the "float" at the start of the line). The file `t1.cc` includes this header file, thereby giving the compiler these three pieces of information it needs to know about `function`.

The other three lines in `function.h` are *code guards*. In brief, they deal with the following scenario: suppose that we have two header files, `A.h` and `B.h`, and that `A.h` includes `B.h`; there are many scenarios in which it makes good sense for a third file, either `.h` or

`.cc`, to include both `A.h` and `B.h`. The code guards ensure that, when all of the includes have been expanded, the compiler sees exactly one copy of `B.h`.

Finally, the file `function.cc` contains the source code for the function named `function`:

```
1  float function ( float i ){
2    return 2.*i;
3  }
```

It names its argument `i`, multiplies this argument by two and returns that value. This code fragment is called the *definition* of the function or the *implementation*(γ) of the function. (The C++ standard uses the word *definition* but *implementation* is in common use.)

We now have a rich enough example to discuss another case in which the same word is frequently used to mean two different things. Sometimes people use the phrase "the source code of the function named `function`" to refer collectively to both `function.h` and `function.cc`; sometimes they use it to refer exclusively to `function.cc`. Unfortunately the only way to distinguish the two uses is from context.

The word *header file* always refers unambiguously to the `.h` file. The term *implementation file* is used to refer unambiguously to the `.cc` file. This name follows from the its contents: it describes how to implement the items declared in the header file.

Based on the above description, when this exercise is run, we expect it to print out:

```
1  a =          3
2  function(a)  6
```

6.4.3 Compile, Link and Run the Exercise

To perform this exercise, first log in and cd to your working directory if you haven't already, then

1. cd to the directory for this exercise and get a directory listing:

   ```
   cd Build/v1
   ```

```
ls
build build2 function.cc function.h t1.cc
```

The two files, `build` and `build2` are scripts that show two different ways to build the code.

2. Compile and link this exercise, then get an updated directory listing:

```
build
```

```
ls
build build2 function.cc function.h
function.o t1 t1.cc t1.o
```

Notice the new files `function.o`, `t1` and `t1.o`.

3. Run the exercise:

```
./t1
a = 3
function(a) 6
```

This matches the expected printout.

Look at the file `build` that you just ran. It has three steps; the first two commands have the -c command line option while the last one does not:

1. It compiles the main program, `t1.cc`, into the object file (with the default name) `t1.o` (which will now be the thing that the term *main program* refers to):
   c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c t1.cc

2. It (separately) compiles `function.cc` into the object file `function.o` (shown in two lines):
   c++ -Wall -Wextra -pedantic -Werror -std=c++11
   -c function.cc

3. It links `t1.o` and `function.o` to form the executable program `t1` (the name of the main program is the argument of the -o option):
   c++ -std=c++11 -o t1 t1.o function.o

You should have noticed that the same command, c++, is used both for compiling and linking. The full story is that when you run the command c++, you are actually running a program that parses its command line to determine which, if any, files need to be compiled and which, if any, files need to be linked. It also determines which of its command line arguments should be forwarded to the compiler and which to the linker. It then runs the compiler and linker as many times as required.

If the -c option is present, it tells c++ to compile only, and not to link. If -c is specified, the `.cc` file(s) to compile must also be specified. Each of the files will be compiled to create its corresponding object file and then processing stops. In our example, the first two commands each compile a single source file. Note that if any `.o` files are given on the command line, c++ will issue a warning and ignore them.

The third command (with no -c option) is the linking step. Even if the -c option is missing, c++ will first look for source files on the command line; if it finds any, it will compile them and put the output into temporary object files. In our example, there are none, so it goes straight to linking. The two just-created object files are specified (at the end, here, but the order is not important); the -o t1 portion of the command tells the linker to write its output (the executable) to the file `t1`.

As it is compiling the main program, `t1.cc`, the compiler recognizes every function that is defined within the file and every function that is called by the code in the file. It recognizes that `t1.cc` defines a function `main()` and that `main()` calls a function named `function`, whose definition is not found inside `t1.cc`. At the point that `t1.cc` calls `function`, the compiler will write to `function` all of the machine code needed to prepare for the call; it will also write all of the machine code needed to use the result of the function. In between these two pieces, the compiler will write machine code that says "call the function whose memory address is" but it must leave an empty placeholder for the address. The placeholder is empty because the compiler does not know the memory address of that function.

The compiler also makes a table that lists all functions defined by the file and all functions that are called by code within the file. The name of each entry in the table is called a *linker*

symbol and the table is called a *symbol table*. When the compiler was compiling `t1.cc` and it found the definition of the main program, it created a linker symbol for the main program and added a notation to say the this file contains the definition of that symbol. When the compiler was compiling `t1.cc` and it encountered the call to `function`, it created a linker symbol for this function; it marked this symbol as an *undefined reference* (because it could not find the definition of `function` within `t1.cc`). The symbol table also lists all of the places in the machine code of `t1.o` that are placeholders that must be updated once the memory address of `function` is known. In this example there is only one such place.

When the compiler writes an object file, it writes out both the compiled code and the table of linker symbols.

In `t1.cc`, the compiled code for the line that begins `std::cout` will do its work by calling a few functions that are found either in the compiler-supplied libraries or the compiler-supplied headers. The linker symbols for the functions found in the libraries will also be listed as undefined references in the symbol table of `t1.o`.

The symbol table in the file `function.o` is simple; it says that this file defines a function named `function` that takes a single argument of type float and that returns a float.

The job of the linker (also invoked by the command `c++`) is to play match-maker. First it inspects the symbol tables inside all of the object files listed on the command line and looks for a linker symbol that defines the location of the main program. If it cannot find one, or if it finds more than one, it will issue an error message and stop. In this example

1. The linker will find the definition of a main program in `t1.o`.
2. It will start to build the executable (output) file by copying the machine code from `t1.o` to the output file.
3. Then it will try to resolve the unresolved references listed in the symbol table of `t1.o`; it does this by looking at the symbol tables of the other object files on the command line. It also knows to look at the symbol tables from a standard set of compiler-supplied and system-supplied libraries.
4. It will discover that `function.o` resolves one of the external references from `t1.o`. So it will copy the machine code from `function.o` to the executable file.

5. It will discover that the the other unresolved references in `t1.o` are found in the compiler-supplied libraries and will copy code from these libraries into the executable.

6. Once all of the machine code has been copied into the executable, the compiler knows the memory address of every function. The compiler can then go into the machine code, find all of the placeholders and update them with the correct memory addresses.

Sometimes resolving one unresolved reference will generate new ones. The linker iterates until (a) all references are resolved and no new unresolved references appear (success) or (b) the same unresolved references continue to appear (error). In the former case, the linker writes the output to the file specified by the -o option; if no -o option is specified the linker will write its output to a file named `a.out`. In the latter case, the linker issues an error message and does not write the output file.

After the link completes, the files `t1.o` and `function.o` are no longer needed because everything that was useful from them was copied into the executable `t1`. You may delete the `.o` files, and the executable will still run.

6.4.4 Alternate Script `build2`

The script `build2` shows an equivalent way of building `t1` that is commonly used for small programs; it does it all on one line. To exercise this script:

1. Stay in the same directory as before, `Build/v1`.

2. Clean up from the previous build and look at the directory contents:

```
rm function.o t1 t1.o
```

```
ls
build build2 function.cc function.h t1.cc
```

3. Run the `build2` script, and again look at directory contents:

```
build2
```

```
ls
build build2 function.cc function.h t1 t1.cc
```

Note that `t1` was created but there are no `.o` files.

4. Execute the program that you just built:

```
./t1
a = 3
function(a) 6
```

Look at the script `build2`; it contains only one command:

```
c++ -Wall -Wextra -pedantic -Werror -std=c++11 -o t1 t1.cc function.cc
```

This script automatically does the same operations as `build` but it knows that the `.o` files are temporaries. Perhaps the command c++ kept the contents of the two `.o` files in memory and never actually wrote them out as disk files. Or, perhaps, the command c++ did explcitly create disk files and deleted them when it was finished. In either case you don't see them when you use `build2`.

6.4.5 Suggested Homework

It takes a bit of experience to decipher the error messages issued by a C++ compiler. The three exercises in this section are intended to introduce you to them so that you (a) get used to looking at them and (b) understand these particular errors if/when you encounter them later.

Each of the following three exercises is independent of the others. Therefore, when you finish with each exercise, you will need to undo the changes you made in the source file(s) before beginning the next exercise.

1. In `Build/v1/t1.cc`, comment out the include directive for `function.h`; rebuild and observe the error message.

2. In `Build/v1/function.cc`, change the return type to `double`; rebuild and observe the error message.

3. In `Build/v1/t1.cc`, change `float a=3.` to `double a=3.`; rebuild and run. This will work without error and will produce the same output as before.

The first homework exercise will issue the diagnostic:

```
t1.cc: In function ‘int main()’:␣\\
t1.cc:10:44:␣error:␣‘function’ was not declared in this scope %}
```

When you see a message like this one, you can guess that either you have not included a required header file or you have misspelled the name of the function.

The second homework exercise will issue the diagnostic (second and last lines split into two here):

```
function.cc: In function ‘double function(float)’:␣\\
function.cc:3:27:␣error:␣new␣declaration␣\\
␣␣␣␣␣␣␣␣␣␣‘double␣function(float)’\\
In file included from function.cc:1:0:\\
function.h:4:7: error: ambiguates old declaration \\
          ‘float function(float)’}
```

This error message says that the compiler has found two functions that have the same signature but different return types. The compiler does not know which of the two functions you want it to use.

The bottom line here is that you must ensure that the definition of a function is consistent with its declaration; and you must ensure that the use of a function is consistent with its declaration.

The third homework exercise illustrates the C++ idea of *automatic type conversion*; in this case the compiler will make a temporary variable of type `float` and set its value to that of `a`:

```
float tmp = a;
```

The compiler will then use this temporary variable as the argument of the function. Consult the standard C++ documentation to understand when automatic type conversions may occur; see Section 6.7.

6.5 C++ Exercise 3: Libraries

Multiple compiled object code files can be grouped into a single file known as a *library*, obviating the need to specify each and every object file when linking; you can reference the libraries instead. This simplifies the multiple use and sharing of software components.

Two Linux C/C++ library types can be created:

- static libraries of object code (filenames for which end in `.a`) that are linked with, and become part of, the application. *art* does not use static libraries.
- dynamically linked libraries (filenames end in `.so` for standard UNIX and in `.dylib` for OS X). Multiple *art* jobs running simultaneously can dynamically load the same copy of a library of this kind instead of making an exclusive copy of it; this substantially reduces the amount of memory needed by a set of jobs using the same libraries.

6.5.1 What You Will Learn

In this section you will repeat the example of Section 6.4 with a variation. You will create an object library, insert `function.o` into that library and use that library in the link step. This pattern generalizes easily to the case that you will encounter in your experiment software, where object libraries will typically contain many object files.

6.5.2 Building and Running the Exercise

To perform this exercise, do the following:

1. Log in and establish your working environment (Section 6.2).
2. cd to your working directory.
3. cd to the directory for this exercise and get a directory listing:

   ```
   cd Libraries/v1
   ```

```
ls
build build2 build3 function.cc function.h
t1.cc
```

The three files, `function.cc`, `function.h` and `t1.cc` are identical to those from the previous exercise. The three files, `build`, `build2` and `build3` are scripts that show three different ways to build the main program in this exercise.

4. Compile and link this exercise using `build`, then compare the directory listing to that taken pre-build:

```
build
```

```
ls
build build3 function.h libpackage1.a t1.cc
build2 function.cc function.o t1 t1.o
```

5. Execute the main program:

```
./t1
a = 3
function(a) 6
```

This matches the expected printout. Now let's look at the script `build`. It has four parts which do the following things:

1. Compiles `function.cc`; the same as the previous exercise:
   c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c function.cc
2. Creates the library named `libpackage1.a` and adds `function.o` to it:
   ar rc libpackage1.a function.o
   Note that the name of the library must come before the name of the object file.
3. Compiles `t1.cc`; the same as the previous exercise:

c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c t1.cc

4. Links the main program against `libpackage1.a` and the system libraries:
   c++ -o t1 t1.o libpackage1.a

The two new features are in step 2, which creates the object library, and step 4, in which `function.o` is replaced in the link list with `libpackage1.a`. If you have many `.o` files to add to the library, you may add them one at a time by repeating step 2 or you may add them all in one command. When you do the latter you may name each object file separately or may use a wildcard:

```
ar rc libpackage1.a *.o
```

In `libpackage1.a` the string `package1` has no special meaning; it was an arbitrary name chosen for this exercise. *Actually it was chosen in anticipation of a future exercise that is not yet written up.*

The other parts of the name, the prefix `lib` and the suffix `.a`, are part of a long-standing Unix convention and some Unix tools presume that object libraries are named following this convention. You should always follow this convention. The use of this convention is illustrated by the scripts `build2` and `build3`.

To perform the exercise using `build2`, stay in the same directory and cleanup then rebuild as follows:

1. Remove files built by `build1`:

   ```
   rm function.o t1.o libpackage1.a t1
   ```

2. Build the code with `build2` and look at the directory contents:

   ```
   build2
   ```

   ```
   ls
   build build3 function.h libpackage1.a t1.cc
   build2 function.cc function.o t1 t1.o
   ```

3. Run `./t1` as before.

The only difference between `build` and `build2` is the link line. The version from `build` is:

```
c++ -o t1 t1.o libpackage1.a
```

while that from `build2` is:

```
c++ -o t1 t1.o -L. -lpackage1
```

In the script `build`, the path to the library, relative or absolute, is written explicitly on the command line. In the script `build2`, two new elements are introduced. The command line may contain any number of -L options; the argument of each option is the name of a directory. The ensemble of all of the -L options forms a search path to look for named libraries; the path is searched in the order in which the -L options appear on the line. The names of libraries are specified with the -l options (this is a lower case letter L, not the numeral one); if a -l option has an argument of XXX (or package1), then the linker with search the path defined by the -L options for a file with the name `libXXX.a` (or `libpackage1.a`).

In the above, the dot in -L. is the usual Unix pathname that denotes the current working directory. And it is important that there be no whitespace after a -L or a -l option and its value.

This syntax generalizes to multiple libraries in multiple directories as follows. Suppose that the libraries `libaaa.a`, `libbbb.a` and `libccc.a` are in the directory `L1` and that the libraries `libddd.a`, `libeee.a` and `libfff.a` are in the directory `L2`. In this case, the link list would look like (split here into two lines):

```
-L<path-to-L1> -laaa -lbbb -lccc
-L<path-to-L2> -lddd -leee -lfff
```

The -L -l syntax is in common use throughout many Unix build systems: if your link list contains many object libraries from a single directory then it is not necessary to repeatedly specify the path to the directory; once is enough. If you are writing link lists by hand, this is very convenient. In a script, if the path name of the directory is very long, this convention makes a much more readable link list.

To perform the exercise using `build3`, stay in the same directory and cleanup then rebuild as follows:

1. Remove files built by `build2`:

   ```
   rm function.o t1.o libpackage1.a t1
   ```

2. Build the code with `build2` and look at the directory contents:

   ```
   build3

   ls
   build build3 function.h libpackage1.a t1.cc
   build2 function.cc function.o t1
   ```

3. Run `./t1` as before

The difference between `build2` and `build3` is that `build3` compiles the main program and links it, all one one line. `build2`, on the other hand did the two steps separately.

6.6 Classes

6.6.1 Introduction

The comments in the sample program used in Section 6.3 emphasized that every variable has a type: `int`, `float`, `std::string`, `std::vector<std::string>`, and so on. One of the basic building blocks of C++ is that users may define their own types; user-defined types may be built-up from all types, including other user-defined types.

The most common user-defined type is called a *class*(γ). As you work through the Workbook exercises, you will see classes that are defined by the Workbook itself; you will also see classes defined by the toyExperiment UPS product; you will see classes defined by *art* and you will see classes defined by the many UPS products that support *art*. You will also write some classes of your own. When you work with the software for your experiment you will work with classes defined within your experiment's software.

In general, a class contains both a *declaration* (what it consists of) and an *implementation*(γ) (what to do with the parts). The declaration contains some data, called *data members* or *member data*, plus some functions (called *member functions*) that will operate on that data, but it is legal for a class declaration (and therefore, a class) to contain only data or only functions. A class *declaration* has the following form:

```
class MyClassName {
    // required: declarations of all members of the class
    // optional: definitions of some members of the class
};
```

The string *class* is an identifier that is reserved to C++ and may not be used for any user-defined *identifiers*. This construct tells the C++ compiler that `MyClassName` is the name of a class; everything that is between the braces is part of the class declaration.

The remainder of Section 6.6 will give many examples of *members* of a class.

In a class declaration, the semi-colon after the closing brace is important.

The upcoming sections will illustrate some features of classes, with an emphasis on features that will be important in the earlier Workbook exercises. This is not indended to be a comprehensive description of classes. To illustrate, we will show nine versions of a class named `Point` that represents a point in a plane. The first version will be simple and each subsequent version will add features.

This documentation will use technically correct language so that you will find it easier to read the standard reference materials.

6.6.2 C++ Exercise 4 v1: The Most Basic Version

Here you will see a very basic version of the class `Point` and an illustration of how `Point` can be used. The ideas of *data members* (sometimes called *member data*), *objects* and *instantiation* will be defined.

To build and run this example:

1. Log in and follow the follow the steps in Section 6.2.

2. cd to the directory for this exercise and examine it:

   ```
   cd Classes/v1/
   ```

   ```
   ls
   Point.h ptest.cc
   ```

   Within the subdirectory `v1` the main program for this exercise is the file `ptest.cc`. The file `Point.h` contains the first version of the class `Point`; shown in the listing on page 70.

3. Build the exercise.

   ```
   ../build
   ```

   ```
   ls
   Point.h ptest ptest.cc
   ```

   The file named `ptest` is the executable program.

4. Run the exercise:

   ```
   ./ptest
   p0: (2.31827e-317, 0)
   p0: (1, 2)
   p1: (3, 4)
   p2: (1, 2)
   Address of p0: 0x7fff883fe680
   Address of p1: 0x7fff883fe670
   Address of p2: 0x7fff883fe660
   ```

The values printed out in the first line of the output may be different when you run the

program (remember initializaion?). When you look at the code you will see that `p0` is not properly initialized and therefore contains stale data. The last three lines of output should also differ when you run the program; they are memory addresses.

Look at the header file `Point.h` which shows the basic version of the class `Point`. The three lines starting with # make up a code guard.

```
#ifndef Point_h
#define Point_h

    class Point {

    public :
      double x;
      double y;
};



#endif /* Point_h *
```

These two lines plus the last line make up a code guard.

The class declaration says that the name of the class is `Point`.

The body of the class declaration (the lines between the braces {...}) declares two data members of the class, named `x` and `y`, both of which are of type `double`. The line `public:` says that the member data `x` and `y` are accessible by any code. Instead of `public`, members may be declared `private` or `protected`; `private` member data will be discussed in Section 6.6.7.

Last line of the code guard

In this exercise there is no file `Point.cc` because the class `Point` consists only of a declaration; there is no implementation to put in a corresponding `.cc` file.

Look at the function `main()` (the *main program*) in `ptest.cc`, below, which illustrates the use of the class `Point`This file includes `Point.h` so that the compiler will know about the class `Point` when it begins execution. It also includes the C++ header `<iostream>` which enables printing with `std::cout`.

```
#include "Point.h"
#include <iostream>

int main()
{
  Point p0;
std::cout << "p0: (" << p0.x << ", " << p0.y << ")" << std::endl;

p0.x = 1.0;
p0.y = 2.0;

std::cout << "p0: (" << p0.x << ", " << p0.y << ")" << std::endl;

Point p1;
p1.x = 3.0;
p1.y = 4.0;
std::cout << "p1: (" << p1.x << ", " << p1.y << ")" << std::endl;

Point p2 = p0;

std::cout << "p2: (" << p2.x << ", " << p2.y << ")" << std::endl;

std::cout << "Address of p0: " << &p0 << std::endl;
std::cout << "Address of p1: " << &p1 << std::endl;
std::cout << "Address of p2: " << &p2 << std::endl;

return 0;
 }
}
```

`Point p0;` declares `p0` the name of a variable whose type is (the class) `Point` then prints out the values of the two data members. In C++, the dot (period) character used this way is called the *member selection operator*.

These lines show how to modify the values of the data members of the object `p0`, then the program makes a printout to verify that the values have indeed changed.

These lines declare another object, named `p1`, of type `Point` and *assign* values to its data members. These are followed by a print statement.

This declares that the object named `p2` is of type `Point` and it assigns the value of `p2` to be a copy of the value of `p0`. When the compiler sees this line, it knows to copy all of the data members of the class; this is a tremendous convenience for classes with many data members. Again, a print statement follows.

The last three lines print the address of each of the three objects, `p0`, `p1` and `p2` in hexidecimal fomat.

When the first line of code in the `main()` program,

```
Point p0;
```

is executed, the program will ensure that memory has been allocated† to hold the data members of `p0`. If the class `Point` contained code to initialize data members then the program would also run that, but `Point` does not have any such code. Therefore the data members take on whatever values happened to preexist in the memory that was allocated for them.

Some other standard pieces of C++ nomenclature can now be defined:

1. The identifier `p0` refers to a variable in the source code whose type is `Point`.
2. When the running program executes this line of code, it *instantiates*(γ) the *object*(γ) with the identifier `p0`.
3. The object with the identifier `p0` is an *instance*(γ) of the class `Point`.
4. The identifier `p0` now also refers to a region of memory containing the bytes belonging to an object of type `Point`.

An important take-away from the above is that a *variable* is an identifier in a source code file while an *object* is something that exists in the computer memory. Most of the time a one-to-one correspondence exists betweeen variables in the source code and objects in memory. There are exceptions, however, for example, sometimes a compiler needs to make anonymous temporary objects that do not correspond to any variable in the source code, and sometimes two or more variables in the source code can refer to the same object in memory.

The last section of the main program (and of `ptest.cc` itself) prints the address of each of the three objects, `p0`, `p1` and `p2`. The addresses are represented in hexadecimal (base 16) format. On almost all computers, the length of a `double` is eight bytes. Therefore an object of type `Point` will have a length of 16 bytes. If you look at the printout made by `ptest` you will see that the addresses of `p0`, `p01` and `p2` are separated by 16 bytes; therefore the three objects are contiguous in memory.

Figure 6.1 shows a diagram of the computer memory at the end of running `ptest`; the outer box (blue outline) represents the memory of the computer; each filled colored box represents one of the three objects in this program. The diagram shows them in contiguous memory locations, which is not necessary; there could have been gaps between the

† This is deliberately vague — there are many ways to allocate memory, and sometimes the memory allocation is actually done much earlier on, perhaps at link time or at load time.

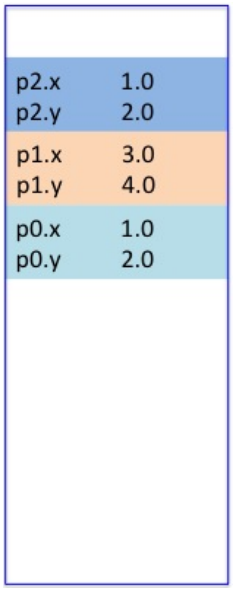


Figure 6.1: Memory diagram at the end of a run of Classes/v1/ptest.cc

memory locations in Figure 6.1.

Now, for a bit more terminology: each of the objects `p0`, `p1` and `p2` has the three attributes required of an *object*:

1. a *state*, given by the values of its data members
2. the ability to have operations performed on it: e.g., setting/reading in value of a data member, assigning value of object of a given type to another of the same type
3. an *identity*: a unique address in memory

6.6.3 C++ Exercise 4 v2: The Default Constructor

This exercise expands the class `Point` by adding a default *constructor*(γ).

To build and run this example:

1. Log in and follow the follow the steps in Section 6.2.

2. Go to the directory for this exercise:

   ```
   cd Classes/v2
   ```

   ```
   ls
   Point.cc Point.h ptest.cc
   ```

   In this example, `Point.cc` is a new file.

3. Build the exercise:

   ```
   ../build
   ```

   ```
   ls
   Point.cc Point.h ptest ptest.cc
   ```

4. Run the exercise:

   ```
   ptest
   p0: (0, 0)
   p0: (3.1, 2.7)
   ```

When you run the code, all of the printout should match the above printout exactly.

Look at `Point.h`. There is one new line in the body of the class declaration:

```
Point();
```

The parentheses tell you that this new member is some sort of function. A C++ class may have several different kinds of functions.

A function that has the same name as the class itself has a special role and is called a *constructor*; if a constructor takes no arguments it is called a *default constructor*. In informal written material, the word constructor is sometimes written as *c'tor*.

`Point.h` declares that the class `Point` has a default constructor, but does not define it (i.e., provide an implementation). The definition/implementation of the constructor is found in the file `Point.cc`.

Look at the file `Point.cc`. It "includes" the header file `Point.h` because the compiler needs to know all about this class before it can compile the code that it finds in `Point.cc`. The rest of the file contains a *definition* of the constructor. The syntax `Point::` says that the function to the right of the `::` is part of (a member of) the class `Point`. The body of the constructor gives initial values to the two data members, `x` and `y`.

Look at the program `ptest.cc`. The first line of the `main` program is again

```
Point p0;
```

When the program executes this line, the first step is the same as before: it ensures that memory has been allocated for the data members of `p0`. This time, however, it also calls the default constructor of the class `Point` (declared in `Point.h`), which initializes the two data members (per `Point.cc`) such that they have well defined initial values. This is reflected in the printout made by the next line.

The next block of the program assigns new values to the data members of `p0` and prints them out.

In the previous example, `Classes/v1/ptest.cc`, a few things happened behind the scenes that will make more sense now that you know what a constructor is.

1. Since the class `Point` did not contain a default constructor , the compiler (implicitly) wrote a default constructor for you ; this default constructor simply "default constructed" each of the data members.
2. The (implicit) constructor of the built-in type `double` did nothing, leaving the data members `x` and `y` uninitialized.

6.6.4 C++ Exercise 4 v3: Constructors with Arguments

This exercise introduces four new ideas:

1. constructors with arguments
2. the copy constructor

3. implicitly generated constructor
4. single-phase construction vs. two-phase construction

To build and run this exercise, cd to the directory `Classes/v3` and follow the same instructions as in Section 6.6.3. When you run the `ptest` program, you should see the following output:

ptest

```
p0: (1, 2)
p1: (1, 2)
```

Look at the file `Point.h`. This contains one new line:

```
Point( double ax, double ay);
```

This line declares a second constructor; we know it is a constructor because it is a function whose name is the same as the name of the class. It is distinguishable from the default constructor because its argument list is different than that of the default constructor. As before, the file `Point.h` contains only the declaration of this constructor, not its *definition* (aka *implementation*).

Look at the file `Point.cc`. The new content in this file is the implementation of the new constructor; it assigns the values of its arguments to the data members. The names of the arguments, `ax` and `ay`, have no meaning to the compiler; they are just identifiers. It is good practice to choose names that bear an obvious relationship to those of the data members. One convention that is sometimes used is to make the name of the argument be the same as that of the data member, but with a prefix lettter `a`, for `a`rgument. Whatever convention you (or your experiment) choose(s), use it consistently. When you update code that was initially written by someone else, follow whatever convention they adopted. Choices of style should be made to reinforce the information present in the code, not to fight it.

Look at the file `ptest.cc`. The first line of the main program is now:

```
Point p0(1.,2.);
```

This line declares the variable `p0` and initializes it by calling the new constructor defined in this section. The next line prints the value of the data members.

The next line of code

```
Point p1(p0);
```

introduces the *copy constructor*. A copy constructor is indicated by code (like the above) that wants to create an exact copy (e.g., `p1`, data-member-for-data-member, of an existing type/class (e.g., `p0`). This exercise does not provide a copy constructor so the compiler implicitly declares a copy constructor, with public access, for that class. (The compiler puts the constructor code directly into the object file; it does not affect the source file.)

In general[‡] if no user-defined constructor exists for a class `A`, the compiler implicitly declares a default, parameterless constructor `A::A()` when it needs to create an object of type `A`. The constructor will have no constructor initializer and a null body.

We recommend that for any class whose data members are either built-in types or simple aggregates of built-in types, of which `Point` is an example, you let the compiler write the copy constructor for you.

If your class has data members that are pointers, or data members that manage some external resource, such as a file that you are writing to, then you will very likely need to write your own copy constructor. There are some other cases in which you should write your own copy constructor, but discussing them here is beyond the scope of this document. When you need to write your own copy constructor, you can learn how to do so from any standard C++ reference; see Section 6.7.

The next line in the file prints the values of the data members of `p1` and you can see that the copy constructor worked as expected.

Notice that in the previous version of `ptest.cc`, the variable `p0` was initialized in three lines:

```
Point p0;
p0.x = 3.1;
p0.y = 2.7;
```

This is called *two-phase construction*. In contrast, the present version uses *single-phase construction* in which the variable `p0` is initialized in one line:

```
Point p0(1.,2.);
```

[‡]Some text in this section is adapted from material in `publib.boulder.ibm.com/infocenter`.

We strongly recommend using single-phase construction whenever possible. Obviously it takes less real estate, but more importantly:

1. Single-phase construction more clearly conveys the intent of the programmer: the intent is to initialize the object `p0`. The second version says this directly. In the first version you needed to do some extra work to recognize that the three lines quoted above formed a logical unit distinct from the remainder of the program. This is not difficult for this simple class, but it can become so with even a little additional complexity.

2. Two-phase construction is less robust. It leaves open the possibility that a future maintainer of the code might not recognize all of the follow-on steps that are part of construction and will use the object before it is fully constructed. This can lead to difficult-to-diagnose run-time errors.

6.6.5 C++ Exercise 4 v4: Colon Initializer Syntax

This version of the class `Point` introduces *colon initializer syntax* for constructors.

To build and run this exercise, cd to the directory `Classes/v4` and follow the same instructions as in the previous two sections. When you run the `ptest` program you should see the following output:

```
ptest
```

```
p0: (1, 2)
p1: (1, 2)
```

The file `Point.h` is unchanged between this version and the previous one.

Now look at the file `Point.cc`, which contains the *definitions* of both constructors. The first thing to look at is the default constructor, which has been rewritten using colon initializer syntax. The rules for the colon-initializer syntax are:

1. A colon must immediately follow the closing parenthesis of the argument list.

2. There must be a comma-separated list of data members, each one initialized by calling one of its constructors.

3. In the initializer list, the data members must be listed in the order in which they appear in the class declaration.
4. The body of the constructor, enclosed in braces, must follow the initializer list.
5. If a data member is missing from the initializer list, its default constructor will be called (constructors for the missing data members will be called in the order in which data members were specified in the class declaration).
6. If no initializer list is present, the compiler will call the default constructor of every data member and it will do so in the order in which data members were specified in the class declaration.

If you think about these rules carefully, you will see that in `Classes/v3/Point.cc`:

1. the compiler did not find an initializer list, so it wrote one that default-constructed `x` and `y`
2. it then wrote the code to make the assignments `x=0` and `y=0`

On the other hand, when the compiler compiled the code for the default constructor in `Classes/v4/Point.cc`, it wrote the code to construct `x` and `y`, both set to zero.

Therefore, the machine code for the `v3` version does more work than that for the `v4` version. In practice `Point` is a sufficiently simple class that the compiler likely recognized and elided all of the unnecessary steps in `v3`; it is likely that the compiler actually produced identical code for the two versions of the class. For a more complex class, however, the compiler may not be able to recognize meaningless extra work and it will write the machine code to do that extra work.

In many cases it does not matter which of these two ways you use to write a constructor; but on those occasions that it does matter, the right answer is always the colon-initializer syntax. So we strongly recommend that you always use the colon initializer syntax. In the Workbook, all classes are written with colon-initializer syntax.

Now look at the second constructor in `Point.cc`; it also uses colon-initializer syntax but it is laid out differently. The difference in layout has no meaning to the compiler — whitespace is whitespace. Choose which ever seems natural to you.

Look at `ptest.cc`. It is the same as the version `v3` and it makes the same printout.

6.6.6 C++ Exercise 4 v5: Member functions

This section will introduce *member functions*(γ), both *const member functions*(γ) and non-const member functions. It will also introduce the header `<cmath>`. Suggested homework for this material follows.

To build and run this exercise, cd to the directory `Classes/v5` and follow the same instructions as in Section 6.6.3. When you run the `ptest` program you should see the following output:

ptest

```
Before p0: (1, 2)  Magnitude: 2.23607  Phi: 1.10715
After  p0: (3, 6)  Magnitude: 6.7082  Phi: 1.10715
```

Look at the file `Point.h`. Compared to version `v4`, this version contains three additional lines:

```
double mag() const;
double phi() const;
void scale( double factor );
```

All three lines declare *member functions*. As the name suggests, a *member function* is a function that can be called and it is a member of the class. Contrast this with a *data member*, such as `x` or `y`, which are not functions. A member function may access any or all of the member data of the class.

The member function named `mag` does not take any arguments and it returns a double; you will see that the value of the double is the magnitude of the 2-vector from the origin to `(x,y)`. The identifier `const` represents a contract between the definition/implementation of `mag` and any code that uses `mag`; it "promises" that the implementation of `mag` will not modify the value of any data members. The consequences of breaking the contract are illustrated in the homework at the end of this subsection.

Similarly, the member function named `phi` takes no arguments, returns a double and has the `const` identifier. You will see that the value of the double is the azimuthal angle of the vector from the origin to the point `(x,y)`.

The third member function, `scale`, takes one argument, `factor`. Its return type is `void`, which means that it returns nothing. You will see that this member function multiplies both

`x` and `y` by `factor` (i.e., changing their values). This function declaration does not have the `const` identifier because it actually does modify member data.

If a member function does not modify any data members, you should always declare it `const` simply as a matter of course. Any negative consequences of not doing so might only become apparent later, at which point a lot of tedious editing will be required to make everything right.

Look at `Point.cc`. Near the top of the file an additional include directive has been added; `<cmath>` is a header from the C++ standard library that declares a set of functions for computing common mathematical operations and transformations. Functions from this library are in the *namespace*(γ) `std`.

Later on in `Point.cc` you will find the definition of `mag`, which computes the magnitude of the 2-vector from the origin to `(x,y)`. To do so, it uses `std::sqrt`, a function declared in the `<cmath>` header that takes the square root of its argument. The identifier `const` that was present in the declaration of `mag` must also be present in its definition.

The next part of `Point.cc` contains the definition of the member function `phi`. To do its work, this member function uses the `atan2` function from the standard library.

The next part of `Point.cc` contains the definition of the member function `scale`. You can see that this member function simply multiplies the two data members by the value of the argument.

The file `ptest.cc` contains a `main()`) program that illustrates these new features. The first line of this function declares and initializes an object, `p0`, of type `Point`. It then prints out the value of its data members, the value returned from calling the function `mag` and the value returned from calling `phi`. This shows how to access a member function: you write the name of the variable, followed by a dot (the *member selection operator*), followed by the name of the member function and its argument list.

The next line calls the member function `scale` with the argument `3`. The printout verifies that the call to `scale` had the intended effect.

One final comment is in order. Many other modern computer languages have ideas very similar to C++ classes and C++ member functions; in some of those languages, the name *method* is the technical term corresponding to *member function* in C++. The name *method*

is not part of the formal definition of C++, but is commonly used nonetheless. In this documentation, the two terms can be considered synonymous.

Here we suggest four activities as homework to help illustrate the meaning of `const` and to familiarize you with the error messages produced by the C++ compiler. Before moving to a subsequent activity, undo the changes that you made in the current activity.

1. In the definition of the member function `Point::mag()`, found in `Point.cc`, before taking the square root, multiply the member datum `x` by 2.

   ```
   double Point::mag() const{
     x *= 2.;
     return std::sqrt( x*x + y*y );
   }
   ```

   Then build the code again; you should see the following diagnostic message:

   ```
   Point.cc: In member function ‘double Point::mag() const’:
   Point.cc:13:8: error: assignment of member ‘Point::x’ in
          read-only object
   ```

2. In `ptest.cc`, change the first line to
   `Point const p0(1,2);`
   Then build the code again; you should see the following diagnostic message:

   ```
   ptest.cc: In function ‘int main()’:
   ptest.cc:13:14: error: no matching function for call to
   ‘Point::scale(double) const’
   ptest.cc:13:14: note: candidate is:
   In file included from ptest.cc:1:0:
   Point.h:13:8: note: void Point::scale(double) <near match>
   Point.h:13:8: note:   no known conversion for implicit
        ‘this’ parameter from ‘const Point*’ to ‘Point*’
   ```

These first two homework exercises illustrate how the compiler enforces the contract defined by the identifier `const` that is present at the end of the declaration of `Point::mag()` and that is absent in the definition of the member function `Point::scale()`. The contract says that the definition of `Point::mag()` may not modify the values of any data members of the class `Point`; users of the class `Point` may count on this behaviour. The contract also says that the definition of the member function `Point::scale()` may modify the values of data members of the class `Point`; users of the class `Point`

must assume that `Point::scale()` will indeed modify member data and act accordingly.[§]

In the first homework exercise, the value of a member datum is modified, thereby breaking the contract. The compiler detects it and issues a diagnostic message.

In the second homework exercise, the variable `p0` is declared `const`; therefore the code may not call non-`const` member functions of `p0`, only `const` member functions. When the compiler sees the call to `p0.mag()` it recognizes that this is a call to `const` member function and compiles the call; when it sees the call to `p0.scale(3.)` it recognizes that this is a call to a non-`const` member function and issues a diagnostic message.

4. In `Point.h`, remove the `const` identifier from the declaration of the member function `Point::mag()`:

   ```
   double mag();
   ```

   Then build the code again; you should see the following diagnostic message:

```
1 Point.cc:12:8: error: prototype for 'double Point::mag()
2       const' does not match any in class 'Point'
3 In file included from Point.cc:1:0:
4 Point.h:11:10: error: candidate is: double Point::mag()
```

5. In `Point.cc`, remove the `const` identifier in definition of the member function `mag`. Then build the code again; you should see the following diagnostic message:

```
1 Point.cc:12:8: error: prototype for 'double Point::mag()'
2 ␣␣␣␣␣␣␣does not match any in class 'Point'
3 In file included from Point.cc:1:0:
4 Point.h:11:10: error: candidate is:
5       double Point::mag() const
```

The third and fourth homework exercises illustrate that the compiler considers two member functions that are identical except for the presence of the `const` identifier to be different functions[¶]. In homework exercise 3, when the compiler tried to compile `Point::mag() const` in `Point.cc`, it looked at the class declaration in `Point.h` and could not find a matching member function declaration; these was a close, but not exact match. Therefore it

[§] C++ has another identifier, `mutable`, that one can use to exempt individual data members from this contract. It's use is beyond the scope of this introduction and it will be described when it is encountered.

[¶] Another way of saying the same thing is that the `const` identifier is part of the *signature*(γ) of a function.

issued a diagnostic message, telling us about the close match, and then stopped. Similarly, in homework exercise 4, it also could not find a match.

6.6.7 C++ Exercise 4 v6: Private Data and Accessor Methods

6.6.7.1 Setters and Getters

This version of the class `Point` is used to illustrate the following ideas:

1. The class `Point` has been redesigned to have private data members with access to them provided by *accessor functions* and *setter functions*.
2. the *this pointer*
3. Even if there are many objects of type `Point` in memory, there is only one copy of the code.

A 2D point class, with member data in Cartesian coordinates, is not a good example of *why* it is often a good idea to have private data. But it does have enough richness to illustrate the mechanics, which is the purpose of this section. Section 6.6.7.3 discusses an example in which having private data makes obvious sense.

To build and run this exercise, cd to the directory `Classes/v6` and follow the same instructions as in Section 6.6.3. When you run the `ptest` program you should see the following output:

```
ptest
```

```
Before p0: (1, 2)  Magnitude: 2.23607  Phi: 1.10715
After  p0: (3, 6)  Magnitude: 6.7082  Phi: 1.10715
p1: (0, 1)  Magnitude: 1  Phi: 1.5708
p1: (1, 0)  Magnitude: 1  Phi: 0
p1: (3, 6)  Magnitude: 6.7082  Phi: 1.10715
```

Look at `Point.h`. Compare it to the version in `v5`:

```
diff -wb Point.h ../v5/
```

Relative to version `v5` the following changes were made:

1. four new member functions have been declared,

(a) `double x() const;`

(b) `double y() const;`

(c) `void set( double ax, double ay);`

(d) `void set( Point const& p);`

2. the data members have been declared private
3. the data members have been renamed from `x` and `y` to `x_` and `y_`

Yes, there are two functions named `set`. Since in C++ the full name of a member function encodes all of the following information:

1. the name of the class it is in
2. the name of the member function
3. the argument list; that is the number, type and order of arguments
4. whether or not the function is `const`

the member functions both named `set` are completely different member functions. As you work through the Workbook you will encounter a lot of this and you should develop the habit of looking at the full function name (i.e., all the parts). The full name of a member function, turned into text string, is called the *mangled name* of the member function; each C++ compiler does this a little differently. All linker symbols related to C++ classes are the mangled names of the members.

If you want to see what mangled names are created for the class `Point`, you can do the following

```
c++ -Wall -Wextra -pedantic -Werror -std=c++11 -c Point.cc
```

```
nm Point.o
```

You can understand the output of nm by reading its man page.

In a class declaration, if any of the identifiers `public`, `private`, or `protected` appear, then all members following that identifier, and before the next such identifier, have the named property. In `Point.h` the two data members are private and all other members are public.

Look at `Point.cc`. Compare it to the version in `v5`:

```
diff -wb Point.cc ../v5/
```

Relative to version `v5` the following changes were made:

1. the data members have been renamed from `x` and `y` to `x_` and `y_`
2. an implementation is present for each of the four new member functions

Inspect the code in the implementation of each of the new member functions. The member function `x()` simply returns the value of the data member `x_`; similarly for the member function `y()`. These are called *accessors*, *accessor functions*, or *getters* [‖] . The notion of *accessor* is often extended to include any member function that returns the value of simple, non-modifying calculations on a subset of the member data; in this sense, the `mag` and `phi` functions of the `Point` class are considered *accessors*.

The two member functions named `set` copy the values of their arguments into the data members of the class. These are, not surprisingly, called *setters* or *setter functions*.

More generally, any member function that modifies the value of any member data is called a *modifier*.

There is no requirement that there be accessors and setters for every data member of a class; indeed, many classes provide no such member functions for many of their data members. If a data member is important for managing internal state but is of no value to a user of the class, then you should certainly not provide an accessor or a setter.

Now that the data members of `Point` are private, i.e., only the code within `Point` is permitted to access these data members directly. All other code must access this information via the accessor and setter functions.

Look at `ptest.cc`. Compare it to the version in `v5`:

```
diff -wb ptest.cc ../v5/
```

Relative to version `v5` the following changes were made:

[‖] There is a coding style in which the function `x()` would have been called something like `GetX()`, `getX()` or `get_x()`; hence the name *getters*. Almost all of the code that you will see in the Workbook omits the `get` in the names of `accessors`; the authors of this code view the `get` as redundant. Within the Workbook, the exception is for accessors defined by ROOT. The Geant4 package also includes the `Get` in the names of its accessors.

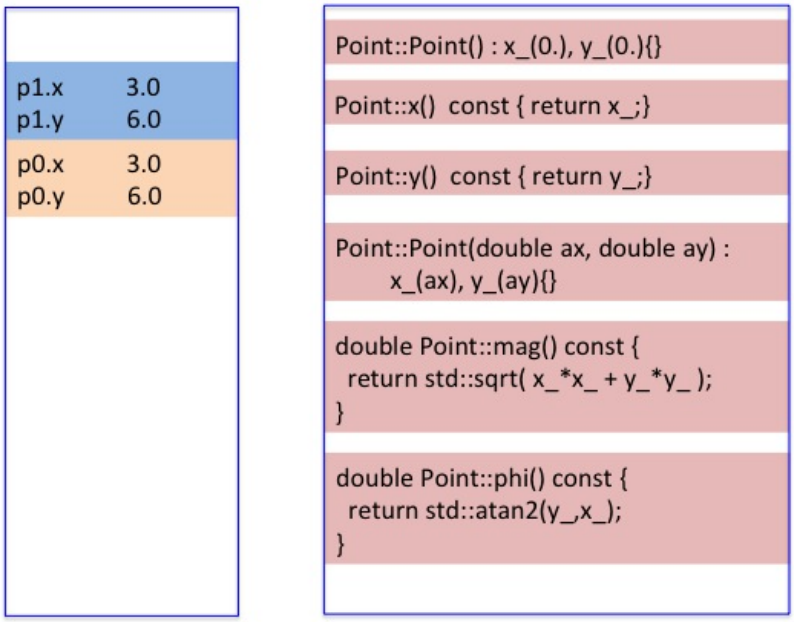


Figure 6.2: Memory diagram at the end of a run of Classes/v6/ptest.cc

1. the printout has been changed to use the accessor functions
2. a new section has been added to illustrate the use of the two set methods

Presumably these are clear.

Figure 6.2 shows a diagram of the computer memory at the end of running this version of `ptest`. The two boxes with the blue outlines represent sections of the computer memory; the part on the left represents that part that is reserved for storing data (such as objects) and the part on the right represents the part of the computer memory that holds the executable code. This is a big oversimplification because, in a real running program, there are many parts of the memory reserved for different sorts of data and many parts reserved for executable code.

The key point in Figure 6.2 is that each object has its own member data but there is only one copy of the code. Even if there are thousands of objects of type `Point`, there will only be one copy of the code. When a line of code asks for `p0.mag()`, the computer will pass the address of `p0` as an argument to the function `mag()`, which will then do its work. When a line of code asks for `p1.mag()`, the computer will pass the address of `p1` as an

argument to the function `mag()`, which will then do its work.

Intially this sounds a little weird: the previous paragraph talks about passing an argument to the function `mag()` but, according to the source code, `mag()` does not take any arguments! The answer is that all member functions have an implied argument that always must be present — the address of the object that the member function will do work on. Because it must always be there, and because the compiler knows that it must always be there, there is no point in actually writing it in the source code! It is by using this so called *hidden argument* that the code for `mag()` knew that `x_` means one thing for `p0` but that it means something else for `p1`.

Every C++ member function has a variable whose name is `this`, which is a pointer to the object on which the member function will do its work. For example, the accessor for `x()` could have been written:

```
double x() const { return this->x_; }
```

This version of the syntax makes it much clearer how there can be one copy of the code even though there are many objects in memory; but it also makes the code harder to read once you have understood how the magic works. There are not many places in which you need to explicitly use the *this* pointer, but there will be some. For further information, consult standard C++ documentation (listed in Section 6.7).

6.6.7.2 What's the deal with the underscore?

C++ will not permit you to use the same name for both a data member and its accessor. Since the accessor is part of the public interface, it should get the simple, obvious, easy-to-type name. Therefore the name of the data member needs to be decorated to make it distinct.

The convention used in the Workbook exercises and in the toyExperiment UPS product is that the names of member data end in an underscore character. There are some other conventions that you may encounter:

```
 _name;
__name;
m_name;
 mName;
```

```
theName;
```

You may also see the choice of a leading underscore, or double underscore, followed by a capital letter. Never do this.

The compiler promises that all of the linker symbols it creates will begin with a leading single or double underscore, followed by a capital letter. Some of the identifiers that you define in a C++ class will be used as part of a linker symbol. If you chose identifiers that match the pattern reserved for symbols created by the compiler there is a chance you will have naming collision with a compiler defined symbol. While this is a very small risk, it seems wise to adopt habits that guarantee that it can never happen.

It is common to extend the pattern for decorating the names of member data to all member data, even those without accessors. One reason for doing so is just symmetry. A second reason has to do with writing member functions; the body of a member function will, in general, use both member data and variables that are local to the member function. If the member data are decorated differently than the local variables, it can make the member functions easier to understand.

6.6.7.3 An example to motivate private data

This section describes a class for which it makes sense to have private data: a 2D point class that has data members `r` and `phi` instead of `x` and `y`. The author of such a class might wish to define a standard representation in which it is guaranteed that `r` be non-negative and that `phi` be on the domain $0 <= \phi < 2\pi$. If the data is public, the class cannot make these guarantees; any code can modify the data members and break the guarantee.

If this class is implemented with private data manipulated by member functions, then the constructors and member functions can enforce the guarantees.

The language used in the software engineering texts is that a guaranteed relationship among the data members is called an *invariant*. If a class has an invariant then the class must have private data.

If a class has no invariant then one is free to choose public data. The Workbook and the toyExperiment never make this choice. One reason is that classes that begin life without an invariant sometimes acquire one as the design matures — we recommend that you plan for this unless you are 100% sure that the class will never have an invariant. A second reason

is that many people who are just starting to learn C++ find it confusing to encounter some classes with private data and others with public data.

6.6.8 C++ Exercise 4 v7: The `inline` Identifier

This section introduces the `inline` identifier.

To build and run this exercise, cd to the directory `Classes/v7` and follow the same instructions as in Section 6.6.3. When you run the `ptest` program you should see the following output:

```
ptest
```

```
p0: ( 1, 2 )  Magnitude: 2.23607  Phi: 1.10715
```

Look at `Point.h` and compare it to the version in `v6`. The new material added to this version is the implementation for the two accessors `x()` and `y()`. These accessors are defined outside of the class declaration.

Look at `Point.cc` and compare it to the version in `v6`. You will see that the implementation of the accessors `x()` and `y()` has been removed.

`Point.h` now contains an almost exact copy of the the implementation of the accessor `x()` that was previously found in the file `Point.cc`; the difference is that it is now preceded by the identifier `inline`. This identifier tells the compiler that it has two options that it may choose from at its discretion.

The first option is that the compiler may decline to write a callable member function `x()`; instead, whenever the member function `x()` is used, the compiler will insert the body of `x()` right into the machine code at that spot. This is called *inlining* the function. For something simple like an accessor, relative to explicitly calling a function, the inlined code is very likely to

1. have a smaller memory footprint
2. execute more quickly

These are both good things.

On the other hand, if you inline a bigger or more complex function, some negative effects

of inlining may appear. If the inlined function is used in many places and if the memory footprint of the inlined code is large compared to the memory footprint of a function call, then the total size of the program can increase. There are various ways in which a large program might run more slowly than a logically equivalent but smaller program. So, if you inline large functions, your program may actually run more slowly!

When the compiler sees the `inline` identifier, it also has a second option: it can choose to ignore it. When the compiler chooses this option it will write many copies of the code for the member function — one copy for each *compilation unit*** in which the function is called. Each compilation unit only knows about its own copy of the function and the compiler calls that copy as needed. The net result is completely negative: the function call is not actually elided so there is no time savings from that; moreover the code has become bigger because there are multiple copies of the function in memory; the larger memory footprint can further slow down execution; and compilation takes longer because multiple copies of the function must be compiled.

C++ does not permit you to force inlining; you may only give a hint to the compiler that a function is appropriate for inlining.

The bottom line is that you should always inline simple accessors and simple setters. Here the adjective *simple* means that they do not do any significant computation and that they do not contain any `if` statements or loops. The decision to inline anything else should only follow careful analysis of information produced by a profiling tool.

Look at the definition of the member function `y()` in `Point.h`. Compared to the definition of the member function `x()` there is a small change in whitespace. This difference is not meaningful to the compiler. You will see several other variations on whitespace when you look at code in the Workbook and its underlying packages.

6.6.9 C++ Exercise 4 v8: Defining Member Functions within the Class Declaration

The version of `Point` in this section introduces the idea that you may provide the definition (implementation) of a member function at the point that it is declared inside the class

** A compilation unit is the unit of code that the compiler considers at one time. For most purposes, each `.cc` file is its own compilation unit.

declaration. This topic is introduced now because you will see this syntax as you work through the Workbook.

To build and run this exercise, cd to the directory `Classes/v8` and follow the same instructions as in Section 6.6.3. When you run the `ptest` program you should see the following output:

ptest

```
p0: ( 1, 2 )  Magnitude: 2.23607  Phi: 1.10715
```

This is the same output made by `v7`.

Look at `Point.h`. The only change relative to `v7` is that the definition of the accessor methods `x()` and `y()` has been moved into the class declaration.

The files `Point.cc` and `ptest.cc` are unchanged with respect to `v7`.

This version of `Point.h` shows that you may define any member function inside the class declaration. When you do this, the `inline` identifier is implicit. Section 6.6.8 discussed some cautions about inappropriate use of inlining; those same cautions apply when a member function is defined inside the class declaration.

When you define a member function within the class declaration, you must not prefix the function name with the class name and the scope resolution operator; that is,

```
double Point::x() const { return x_; }
```

would produce a compiler diagnostic.

In summary, there are two ways to write inlined definitions of member functions. In most cases, the two are entirely equivalent and the choice is simply a matter of style. The one exception occurs when you are writing a class that will become part of an *art* data product, in which case it is recommended that you write the definitions of member functions *outside* of the class declaration.

When writing an *art* data product, the code inside that header file is parsed by software that determines how to write objects of that type to the output disk files and how to read objects of that type from input disk files. The software that does the parsing has some limitations and we need to work around them. The work arounds are easiest if any member functions definitions in the header file are placed outside of the class declarations. For details see

https://cdcvs.fnal.gov/redmine/projects/art/wiki/Data_Product_Design_Guide #Issues-mostly-related-to-ROOT

6.6.10 C++ Exercise 4 v9: The stream insertion operator

The version of `Point` in this section illustrates how to write a *stream insertion operator*. This is the piece of code that lets you print an object without having to print each data member by hand, for example:

```
Point p0(1,2);
std::cout << p0 << std::endl;
```

To build and run this exercise, cd to the directory `Classes/v9` and follow the same instructions as in Section 6.6.3. When you run the `ptest` program you should see the following output:

ptest

```
p0: ( 1, 2 )  Magnitude: 2.23607  Phi: 1.10715
```

This is the same output made by `v7` and `v8`.

Look at `Point.h`. The changes relative to `v7` are the following two additions:

1. an include directive for the header `<iosfwd>`
2. a declaration for the stream insertion operator

Look at `Point.cc`. The changes relative to `v7` are the following two additions:

1. an include directive for the header `<iostream>`
2. the definition of the stream insertion operator.

Look at `ptest.cc`. The only change relative to `v7` is that the printout now uses the stream insertion operator for `p0` instead of inserting each data member of `p0` by hand.

In `Point.h`, the stream insertion operator is declared as (shown here on two lines)

```
std::ostream& operator<<
(std::ostream& ost, Point const& p );
```

If the class whose type is used as second argument is declared in a namespace, then the stream insertion operator must be declared in the same namespace.

When the compiler sees a « operator that has an object of type `std::ostream` on its left hand side and an object of type `Point` on its right hand side, then the compiler will look for a function named `operator«` whose first argument is of type `std::ostream&` and whose second argument is of type `Point const&`. If it finds such a function it will call that function to do the work; if it cannot find such a function it will issue a compiler diagnostic.

The reason that the function returns a `std::ostream&` is that this is the C++ convention that permits us to chain together multiple instances of the « operator:

```
Point p0(1,2), p1(3,4);
std::cout << p0 << `` `` << p1 << std::endl;
```

The C++ compiler parses this left to right. First it recognizes:

```
std::cout << p0;
```

and calls our stream insertion operator to do this work. Then it thinks of the rest of the line as:

```
std::cout << `` `` << p1 << std::endl;
```

Now it recognizes,

```
std::cout << `` ``;
```

and calls the appropriate stream insertion operator to do that work. And so on.

Look at the implementation of the stream insertion operator in `Point.cc`. The first argument, `ost`, is a reference to an object of type output stream; the name `ost` has no meaning to C++; it is just a variable. When writing this operator we don't know and don't care what the output stream is connected to; perhaps it is a file; perhaps it is standard output. In any case, you send output to `ost` just as you do to `std::cout`, which is just another object of type `std::ostream`. In this example we chose to enclose the values of `x_` and `y_` in parentheses and to separate them with a comma; this is simply our choice, not something required by C++ or by *art*.

In this example, the stream insertion operator does *not* end by inserting a newline into `ost`. This is a very common choice as it allows the user of the operator to have full control about line breaks. For a class whose printout is very long and covers many lines, you might decide that this operator should end by inserting newline character; it's your choice.

If you wish to write a stream insertion operator for another class, just follow the pattern used here.

If you want to understand more about why the operator is written the way that it is, consult the standard C++ references; see Section 6.7.

The stream insertion operator is a *free function*(γ), not a member function of the class `Point`; the tie to the class `Point` is via its second argument. Because this function is a free function, it could have been declared in its own header file and its implementation could be provided in its own `.cc` file. However that is not common practice. Instead the common practice is as shown in this example: to include it in `Point.h` and `Point.cc`.

The choice of whether or not to put the declaration of the stream insertion operator into its own header file is a tradeoff between the following two criteria:

1. it is convenient to have it there; otherwise you would have to remember to include an additional header file when you want to use this operator

2. one can imagine many simple free functions that take an object of type `Point` as an argument. If we put them all inside `Point.h`, and if they are only infrequently used, then the compiler will waste time processing those declarations every time `Point.h` is included somewhere.

Ultimately this is a judgement call and the code in this example follows the recommendations made by the *art* development team. Their recommendation is that the following sorts of free functions, and only these sorts, should be included in header files containing a class declaration:

1. the stream insertion operator for that class

2. out of class arithmetic and comparison operators

With one exception, if including a function declaration in `Point.h` requires the inclusion of an additional header in `Point.h`, declare that function in a different header file. The exception is that it is okay to include `<iosfwd>`.

6.6.11 Review

The class `Point` is an example of a class that is primarily concerned with providing convenient access to the data it contains. Not all classes are like this; when you work through the Workbook, you will write some classes that are primarily concerned with packaging convenient access to a set of related functions.

1. class
2. object
3. identifier
4. free function
5. member function

6.7 C++ References

This section lists some recommended C++ references, both text books and online materials.

The following references describe the C++ core language,

- Stroustrup, Bjarne: "The C++ Programming Language, Special Third Edition", Addison-Wesley, 2000. ISBN 0-201-70073-5.
- http://www.cplusplus.com/doc/tutorial/

The following references describe the C++ Standard Library,

- Josuttis, Nicolai M., "The C++ Standard Library: Tutorial and Reference", Addison-Wesley, 1999. ISBN 0-201-37926-0.
- http://www.cplusplus.com/reference

The following contains an introductory tutorial. Many copies of this book are available at the Fermilab library. It is a very good introduction to the big ideas of C++ and Object Oriented Programming but it is not a fast entry point to the C++ skills needed for HEP.

- Andrew Koenig and Barbara E. Moo, "Accelerated C++: Practical Programming by Example" Addison-Wesley, 2000. ISBN 0-201-70353-X.

The following contains a discussion of recommended best practices,

- Herb Sutter and Andrei Alexandrescu, "C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.", Addison-Wesley, 2005. ISBN 0-321-11358-6.

7 Using External Products in UPS

Section 3.6.8 introduced the idea of external products. For the Intensity Frontier experiments (and for Fermilab-based experiments in general), access to external products is provided by a Fermilab-developed product-management package called Unix Product Support (UPS). An important UPS feature – demanded by most experiments as their code evolves – is its support for multiple versions of a product and multiple builds (e.g., for different platforms) per version.

Another notable feature is its capacity to handle multiple databases of products. So, for example, on Fermilab computers, login scripts (see Section 4.9) set up the UPS system, providing access to a database of products commonly used at Fermilab.

The *art* Workbook and your experiment's code will require additional products (available in other databases). For example, each experiment will provide a copy of the toyExperiment product in its experiment-specific UPS database.

In this chapter you will learn how to see which products UPS makes available, how UPS handles variants of a given product, how you use UPS to initialize a product provided in one of its databases and about the environment variables that UPS defines.

7.1 The UPS Database List: PRODUCTS

The act of setting up UPS defines a number of environment variables (discussed in Section 7.5), one of which is PRODUCTS. This particularly important environment variable merits its own section.

The environment variable PRODUCTS is a colon-delimited list of directory names, i.e., it is a path (see Section 4.6). Each directory in PRODUCTS is the name of a *UPS database*,

meaning simply that each directory functions as a repository of information about one or more products. When UPS looks for a product, it checks each directory in PRODUCTS, in the order listed, and takes the first match.

If you are on a Fermilab machine, you can look at the value of PRODUCTS just after logging in, before sourcing your site-specific setup script. Run printenv:

```
printenv PRODUCTS
```

It should have a value of

```
/grid/fermiapp/products/common/db
```

This generic Fermilab UPS database contains a handful of software products commonly used at Fermilab; most of these products are used by all of the Intensity Frontier Experiments. This database does not contain any of the experiment-specific software nor does it contain products such as *ROOT*(γ), *Geant4*(γ), CLHEP or *art*. While these last few products are indeed used by multiple experiments, they are often custom-built for each experiment and as such are distributed via the experiment-specific (i.e., separate) UPS databases.

After you source your site-specific setup script, look at PRODUCTS again. It will probably contain multiple directories, thus making many more products available in your "site" environment. For example, on the DS50+Fermilab site, after running the DS50 setup script, PRODUCTS contains:

```
/ds50/app/products/:grid/fermiapp/products/common/db
```

You can see which products PRODUCTS contains by running ls on its directories, one-by-one, e.g.,

```
ls /grid/fermiapp/products/common/db
```

```
afs   git      ifdhc         mu2e             python          ...
cpn   gitflow  jobsub_tools  oracle_tnsnames  ...
encp  gits     login         perl             setpath         ...
```

```
ls /ds50/app/products
```

```
art               cetpkgsupport  g4neutronxs  libxml2          ...
artdaq            clhep          g4nucleonxs  messagefacility  ...
```

```
art_suite           cmake          g4photon     mpich            ...
art_workbook_base   cpp0x          g4pii        mvapich2         ...
boost               cppunit        g4radiative  python           ...
caencomm            ds50daq        g4surface    root             ...
...
```

Each directory name in these listings corresponds to the name of a UPS product. If you are on a different experiment, the precise contents of your experiment's product directory may be slightly different. Among other things, both databases contain a subdirectory named `ups`*; this is for the UPS system itself. In this sense, all these products, including *art*, toyExperiment and even the product(s) containing your experiment's code, regard UPS as just another external product.

7.2 UPS Handling of Variants of a Product

An important feature of UPS is its capacity to make multiple variants of a product available to users. This of course includes different versions, but beyond that, a given version of a product may be built more than one way, e.g., for use by different operating systems (what UPS distinguishes as *flavors*). For example, a product might be built once for use with SLF5 and again for use with SLF6. A product may be built with different versions of the C++ compiler, e.g., with the production version and with a version under test. A product may be built with full compiler optimization or with the maximum debugging features enabled. Many variants can exist. UPS provides a way to select a particular build via an idea named *qualifiers*.

The full identifier of a UPS product includes its product name, its version, its flavor and its full set of qualifiers. In Section 7.3, you will see how to fully identify a product when you set it up.

7.3 The setup Command: Syntax and Function

Any given UPS database contains several to many, many products. To select a product and make it available for use, you use the setup command.

*`ups` appears in both listings; as always, the first match wins!

In most cases the correct flavor can be automatically detected by setup and need not be specified. However, if needed, flavor, in addition to various qualifiers and options can be specified. These are listed in the UPS documentation referenced later in this section. The version, if specified, must directly follow the product name in the command line, e.g.,:

```
setup <options> <product-name> <product-version> -f <flavor>  -q <qualifiers>
```

Putting in real-looking values, it would look something like:

```
setup -R myproduct v3_2 -f SLF5 -q BUILD_A
```

What does the setup command actually do? It may do any or all of the following:

- define some environment variables
- define some bash functions
- define some aliases
- add elements to your PATH
- setup additional products on which it depends

Setting up dependent products works recursively. In this way, a single setup command may trigger the setup of, say, 15 or 20 products.

When you follow a given site-specific setup procedure, the PRODUCTS environment variable will be extended to include your experiment-specific UPS repository.

setup is a bash function (defined by the UPS product when it was initialized) that shadows a Unix system-configuration command also named setup, usually found in `/usr/bin/setup` or `/usr/sbin/setup`. Running the right 'setup' should work automatically as long as UPS is properly initialized. If it's not, setup returns the error message:

```
You are attempting to run ``setup'' which requires administrative
privileges, but more information is needed in order to do so.
```

If this happens, the simplest solution is to log out and log in again. Make sure that you carefully follow the instructions for doing the site specific setup procedure.

Few people will need to know more than the above about the UPS system. Those who do can consult the full UPS documentation at:

http://www.fnal.gov/docs/products/ups/ReferenceManual/index.html

7.4 Current Versions of Products

For some UPS products, but not all, the site administrator may define a particular fully-qualified version of the product as the default version. In the language of UPS this notion of default is called the *current* version. If a current version has been defined for a product, you can set up that product with the command:

```
setup <product-name>
```

When you run this, the UPS system will automatically insert the version and qualifiers of the version that has been declared current.

Having a current version is a handy feature for products that add convenience features to your interactive environment; as improvements are added, you automatically get them.

However the notion of a current version is very dangerous if you want to ensure that software built at one site will build in exactly the same way on all other sites. For this reason, the Workbook fully specifies the version number and qualifiers of all products that it requires; and in turn, the products used by the Workbook make fully qualified requests for the products on which they depend.

7.5 Environment Variables Defined by UPS

When your login script or site-specific setup script initializes UPS, it defines many environment variables in addition to PRODUCTS (Section 7.1), one of which is UPS_DIR, the root directory of the currently selected version of UPS. The script also adds `$UPS_DIR/bin` to your PATH, which makes some UPS-related commands visible to your shell. Finally, it defines the bash function setup (see Sections 4.8 and 7.3). When you use the setup command, as illustrated below, it is this bash function that does the work.

In discussing the other important variables, the toyExperiment product will be used as an example product. For a different product, you would replace "toyExperiment" or "TOYEXPERIMENT" in the following text by the product's name. Once you have followed your appropriate setup procedure (Table 5.1) you can issue the following command this

is informational for the purposes of this section; you don't need to do it until you start running the first Workbook exercise):

```
setup toyExperiment v0_00_15 -qe2:prof
```

The version and qualifiers shown here are the ones to use for the Workbook exercises. When the setup command returns, the following environment variables will be defined:

TOYEXPERIMENT_DIR defines the root DIRectory of the chosen UPS product

TOYEXPERIMENT_INC defines the path to the root directory of the C++ header files that are provided by this product (so called because the header files are INCluded)

TOYEXPERIMENT_LIB defines the directory that contains all of the dynamic object LIBraries (ending in `.so`) that are provided by this product

Almost all UPS products that you will use in the Workbook define these three environment variables. Several, including toyExperiment, define many more. Once you're running the exercises, you will be able to see all of the environment variables defined by the toyExperiment product by issuing the following command:

```
printenv | grep TOYEXPERIMENT
```

Many software products have version numbers that contain dot characters. UPS requires that version numbers not contain any dot characters; by convention, version dots are replaced with underscores. Therefore v0.00.14 becomes v0_00_14. Also by convention, the environment variables are all upper case, regardless of the case used in the product names.

7.6 Finding Header Files

7.6.1 Introduction

Header files were introduced in Section 6.3.2. Recall that a header file typically contains the "parts list" for its associated `.cc` source file and is "included" in the `.cc` file.

The software for the Workbook depends on a large number of external products; the same is true, on an even larger scale, for the software in your experiment. The preceeding sec-

tions in this chapter discussed how to establish a working environment in which all of these software products are available for use.

When you are working with the code in the Workbook, and when you are working on your experiment, you will frequently encounter C++ classes and functions that come from these external products. An important skill is to be able to identify them when you see them and to be able to follow the clues back to their source and documentation. This section will describe how to do that.

An important aid to finding documentation is the use of *namespaces*; if you are not familiar with namespaces, consult the standard C++ documentation.

7.6.2 Finding *art* Header Files

This subsection will use the example of the class `art::Event` to illustrate how to find header files from the *art* UPS product; this will serve as a model for finding header files from most other UPS products.

The class that holds the *art* abstraction of an HEP event is named, `art::Event`; that is, the class `Event` is in the namespace `art`. In fact, all classes and functions defined by *art* are in the namespace `art`. The primary reason for this is to minimize the chances of accidental name collisions between *art* and other codes; but it also serves a very useful documentation role and is one of the clues you can use to find header files.

If you look at code that uses `art::Event` you will almost always find that the file includes the following header file:

```
#include "art/Framework/Principal/Event.h"
```

The *art* UPS product has been designed so that the relative path used to include any *art* header file starts with the directory `art`; this is another clue that the class or function of interest is part of *art*.

When you setup the *art* UPS product, it defines the environment variable ART_INC, which points to the root of the header file tree for *art*. You now have enough information to discover where to find the header file for `art::Event`; it is at

```
$ART_INC/art/Framework/Principal/Event.h
```

You can follow this same pattern for any class or function that is part of *art*. This will only work if you are in an environment in which ART_INC has been defined, which will be described in Chapters 9 and 10.

If you are new to C++, you will likely find this header file difficult to understand; you do not need to understand it when you first encounter it but, for future reference, you do need to know where to find it.

Earlier in this section, you read that if a C++ file uses `art::Event`, it would *almost always* include the appropriate header file. Why *almost* always? Because the header file `Event.h` might already be included within one of the other headers that are included in your file. If `Event.h` is indirectly included in this way, it does not hurt also to include it explicitly, but it is not required that you do so.[†]

We can summarize this discussion as follows: if a C++ source file uses `art::Event` it must always include the appropriate header file, either directly or indirectly.

art does not rigorously follow the pattern that the name of file is the same as the name of the class or function that it defines. The reason is that some files define multiple classes or functions; in most such cases the file is named after the most important class that it defines.

Finally, from time to time, you will need to dig through several layers of header files to find the information you need.

There are two code browsing tools that you can use to help navigate the layering of header files and to help find class declarations that are not in a file named for the class:

1. use the *art redmine*(γ) repository browser:
   https://cdcvs.fnal.gov/redmine/projects/art/repository/revisions/master/show/art

2. use the LXR code browser: http://cdcvs.fnal.gov/lxr/art/

(In the above, both URLs are live links.)

[†] Actually there is small price to pay for redundant includes; it makes the compiler do unnecessary work, and therefore slows it down. But providing some redundant includes as a pedagodical tool is often a good trade-off; the Workbook will frequently do this.

7.6.3 Finding Headers from Other UPS Products

Section 3.7 introduced the idea that the Workbook is built around a UPS product named toyExperiment, which describes a made-up experiment. All classes and functions defined in this UPS product are defined in the namespace `tex`, which is an acronym-like shorthand for toyExperiment (ToyEXperiment). (This shorthand makes it (a) easier to focus on the name of each class or function rather than the namespace and (b) quicker to type.)

One of the classes from the toyExperiment UPS product is `tex::GenParticle`, which describes particles created by the event generator, the first part of the simulation chain (see Section 3.7.2). The include directive for this class looks like

```
#include "toyExperiment/MCDataProducts/GenParticle.h"
```

As for headers included from *art*, the first element in the relative path to the included file is the name of the UPS product in which it is found. Similarly to *art*, the header file can be found using the environment variable TOYEXPERIMENT_INC:

```
$TOYEXPERIMENT_INC/toyExperiment/MCDataProducts/GenParticle.h
```

With a few exceptions, discussed in Section 7.6.4, if a class or function from a UPS product is used in the Workbook code, it will obey the following pattern:

1. The class will be in a namespace that is unique to the UPS product; the name of the namespace may be the full product name or a shortened version of it.
2. The lead element of the path specified in the include directive will be the name of the UPS product.
3. The UPS product setup command will define an environment variable named <PRODUCT-NAME>_INC, where <PRODUCT-NAME> is in all capital letters.

Using this information, the name of the header file will always be

```
$<PRODUCT-NAME>_INC/<path-specified-in-the-include-directive>
```

This pattern holds for all of the UPS products listed in Table 7.1.

A table listing git- and LXR-based code browsers for many of these UPS products can be found near the top of the web page:
https://cdcvs.fnal.gov/redmine/projects/art/wiki

Table 7.1: For selected UPS Products, this table gives the names of the associated namespaces. The UPS products that do not use namespaces are discussed in Section 7.6.4. ‡The namespace `tex` is also used by the *art* Workbook, which is not a UPS product.

UPS Product	Namespace
art	`art`
boost	`boost`
cet	`cetlib`
clhep	`CLHEP`
fhiclcpp	`fhicl`
messagefacility	`mf`
toyExperiment	`tex`‡

7.6.4 Exceptions: The Workbook, ROOT and Geant4

There are three exceptions to the pattern described in Section 7.6.3:

- the Workbook itself
- ROOT
- Geant4

The Workbook is so tightly coupled to the toyExperiment UPS product that all classes in the Workbook are also in its namespace, `tex`. Note, however, that classes from the Workbook and the toyExperiment UPS product can still be distinguished by the leading element of the relative path found in the include directives for their header files:

- `art-workbook` for the Workbook
- `toyExperiment` for the toyExperiment

The ROOT package is a CERN-supplied software package that is used by *art* to write data to disk files and to read it from disk files. It also provides many data analysis and data presentation tools that are widely used by the HEP community. Major design decisions for ROOT were frozen before namespaces were a stable part of the C++ language, therefore ROOT does not use namespaces. Instead ROOT adopts the following conventions:

1. All class names by defined by ROOT start with the capital letter `T` followed by another upper case letter; for example, `TFile`, `TH1F`, and `TCanvas`.

2. With very few exceptions, all header files defined by ROOT also start with the same pattern; for example, `TFile.h`, `TH1F.h`, and `TCanvas.h`.
3. The names of all global objects defined by ROOT start with a lower case letter `g` followed by an upper case letter; for example `gDirectory`, `gPad` and `gFile`.

The rule for writing an include directive for a header file from ROOT is to write its name without any leading path elements:

```
#include "TFile.h"
```

All of the ROOT header files are found in the directory that is pointed to by the environment variable $ROOT_INC. For example, to see the contents of this file you could enter:

```
less $ROOT_INC/TFile.h
```

Or you can the learn about this class using the reference manual at the CERN web site: http://root.cern.ch/root/html534/ClassIndex.html

You will not see theGeant4 package in the Workbook but it will be used by the software for your experiment, so it is described here for completeness. Geant4 is a toolkit for modeling the propagation particles in electromagnetic fields and for modeling the interactions of particles with matter; it is the core of all detector simulation codes in HEP and is also widely used in both the Medical Imaging community and the Particle Astrophysics community.

As with ROOT, Geant4 was designed before namespaces were a stable part of the C++ language. Therefore Geant4 adopted the following conventions.

1. The names of all identifiers begin with `G4`; for example, `G4Step` and `G4Track`.
2. All header file names defined by Geant4 begin with `G4`; for example, `G4Step.h` and `G4Track.h`.

Most of the header files defined by Geant4 are found in a single directory, which is pointed to by the environment variable G4INCLUDE.

The rule for writing an include directive for a header file from Geant4 is to write its name without any leading path elements:

```
#include "G4Step.h"
```

The workbook does not set up a version of Geant4; therefore G4INCLUDE is not defined. If it were, you would look at this file by:

```
less $G4INCLUDE/G4Step.h
```

Both ROOT and Geant4 define many thousands of classes, functions and global variables. In order to avoid collisions with these identifiers, do not define any identifiers that begin with any of (case-sensitive):

- `T`, followed by an upper case letter
- `g`, followed by an upper case letter
- `G4`

Part II

Workbook

8 Preparation for Running the Workbook Exercises

8.1 Introduction

You will run the Workbook exercises on a computer that is maintained by your experiment, either at Fermilab or at another institution. Many details of the working environment change from site to site* and these differences are parameterized so that (a) it is easy to establish the required environment, and (b) the Workbook exercises work the same way at all sites. In this chapter you will learn how to find and log into the right machine remotely from your local machine (laptop or desktop), and make sure it can support your Workbook work.

Note that it will be possible to install the Workbook software on your local (Unix-like) machine; but this option is not yet ready. When instructions are available, a link to them will be provided here. The instructions in this document will work whether the Workbook code is installed locally or on a remote machine.

8.2 Getting Computer Accounts on Workbook-enabled Machines

In order to run the exercises in the Workbook, you will need an account on a machine that can access your site's installation of the Workbook code. The experiments provide instructions for getting computer accounts on their machines (and various other information for

*Remember, a *site* refers to a unique combination of experiment and institution.

Table 8.1: Experiment-specific information for new users (pages are under `https://cdcvs.fnal.gov/redmine/projects/` except for Mu2e and NOvA)

Experiment	Page for New Users
ArgoNeut	larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
Darkside	darkside-public/wiki/Before_You_Arrive
LArSoft	larsoftsvn
LBNE	larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
MicroBoone	larsoftsvn/wiki/Using_LArSoft_on_the_GPVM_nodes
Muon g-2	g-2/wiki/NewGm2Person
Mu2e	http://mu2e.fnal.gov/atwork/general/userinfo/index.shtml#comp
NOvA	http://www-nova.fnal.gov/NOvA_Collaboration_Information/index.html

new users) on web pages that they maintain, as listed in Table 8.1. The URLs in the table are live hyperlinks.

Currently, each of the experiments using *art* has installed the Workbook code on one of its experiment machines in the Fermilab General Purpose Computing Farm (GPCF).

At time of writing, the new-user instructions for all LArSoft-based experiments are at the LArSoft site; there are no separate instructions for each experiment.

If you would like a computer account on a Fermilab computer in order to evaluate *art*, contact the *art* team (see Section 3.4).

8.3 Choosing a Machine and Logging In

The experiment-specific machines confirmed to host the Workbook code are listed in Table 8.2 In most cases the name given is not the name of an actual computer, but rather a round-robin alias for a cluster. For example, if you log into `mu2evm`, you will actually be connected to one of the five computers `mu2egpvm01` through `mu2egpvm05`. These Mu2e machines share all disks that are relevant to the Workbook exercises, so if you need to log in multiple times, it is perfectly OK if you are logged into two different machines; you will still see all of the same files.

Each experiment's web page has instructions on how to log in to its computers from your local machine.

Table 8.2: Login machines for running the Workbook exercises

Experiment	Name of Login Node
ArgoNeut	`argoneutvm.fnal.gov`
Darkside	`ds50.fnal.gov`
LBNE	`lbnevm.fnal.gov`
MicroBoone	`uboonevm.fnal.gov`
Muon g-2	`gm2gpvm.fnal.gov`
Mu2e	`mu2evm.fnal.gov`
NOνA	`nova-offline.fnal.gov`

8.4 Launching new Windows: Verify X Connectivity

Some of the Workbook exercises will launch an X window from the remote machine that opens in your local machine. To test that this works, type xterm &:

```
xterm &
```

This should, without any messages, give you a new command prompt. After a few seconds, a new shell window should appear on your laptop screen; if you are logging into a Fermilab computer from a remote site, this may take up to 10 seconds. If the window does not appear, or if the command issues an error message, contact a computing expert on your experiment.

To close the new window, type exit at the command prompt in the new window:

```
exit
```

If you have a problem with xterm, it could be a problem with your Kerberos and/or ssh configurations. Try logging in again with ssh -Y.

8.5 Choose an Editor

As you work through the Workbook exericses you will need to edit files. Familiarize yourself with one of the editors available on the computer that is hosting the Workbook. Most Fermilab computers offer four reasonable choices: emacs, vi, vim and nedit. Of these, nedit is probably the most intuitive and user-friendly. All are very powerful once you have learned to use them. Most other sites offer at least the first three choices. You can always

contact your local system administrator to suggest that other editors be installed.

A future version of this documentation suite will include recommended configurations for each editor and will provide links to documentation for each editor.

9 Exercise 1: Running Pre-built *art* Modules

9.1 Introduction

In this first exercise of the Workbook, you will be introduced to the *FHiCL*(γ) configuration language and you will run *art* on several modules that are distributed as part of the toyExperiment UPS product. You will not compile or link any code.

9.2 Prerequisites

Before running any of the exercises in this Workbook, you need to be familiar enough with the material discussed in Part I (Introduction) of this documentation set and Chapter 8 to be able to find information as needed.

If you are following the instructions below on a Mac computer, and if you are reading the instructions from a PDF file, be aware that if you use the mouse or trackpad to cut and paste text from the PDF file into your terminal window, the underscore characters will be turned into spaces. You will have to fix them before the commands will work.

9.3 What You Will Learn

In this exercise you will learn:

- how to use the site-specific setup procedure, which you must do once at the start of each login session.
- a little bit about the *art* run-time environment (Section 9.4)

- how to set up the toyExperiment UPS product (Section 9.6.1)
- how to run an *art* job (Section 9.6.1)
- how to control the number of events to process (Section 9.8.4)
- how to select different input files (Section 9.8.5)
- how to start at a run, subRun or event that is not the first one in the file (Section 9.8.6)
- how to concatenate input files (Section 9.8.5)
- how to write an output file (Section 9.8.9)
- some basics about the grammar and structure of a FHiCL file (Section 9.8)
- how *art* finds modules and configuration (FHiCL) files. (Sections 9.10 and 9.11)

9.4 The *art* Run-time Environment

This discussion is aimed to help you understand the process described in this chapter as a whole and how the pieces fit together in the *art* run-time environment. This environment is summarized in Figure 9.1. In this figure the boxes refer either to locations in memory or to files on a disk.

At the center of the figure is a box labelled "*art* executable;" this represents the *art* main program resident in memory after being loaded. When the *art* executable starts up, it reads its run-time configuration (FHiCL) file, represented by the box to its left. Following instructions from the configuration file, *art* will load dynamic libraries from toyExperiment, from *art*, from ROOT, from CLHEP and from other UPS products. All of these dynamic libraries (`.so` or `.dylib` files) will be found in the appropriate UPS products in LD_LIBRARY_PATH (DYLD_LIBRARY_PATH for OS X), which points to directories in the UPS products area (box at upper right). Also following instructions from the FHiCL file, *art* will look for input files (box labeled "Event-data input files" at right). The FHiCL file will tell *art* to write its event-data and histogram output files to a particular directory (box at lower right).

One remaining box in the figure (at right, second from bottom) is not encountered in the first Workbook exercise but has been provided for completeness. In most *art* jobs it is necessary to access experiment-related geometry and conditions information; in a mature

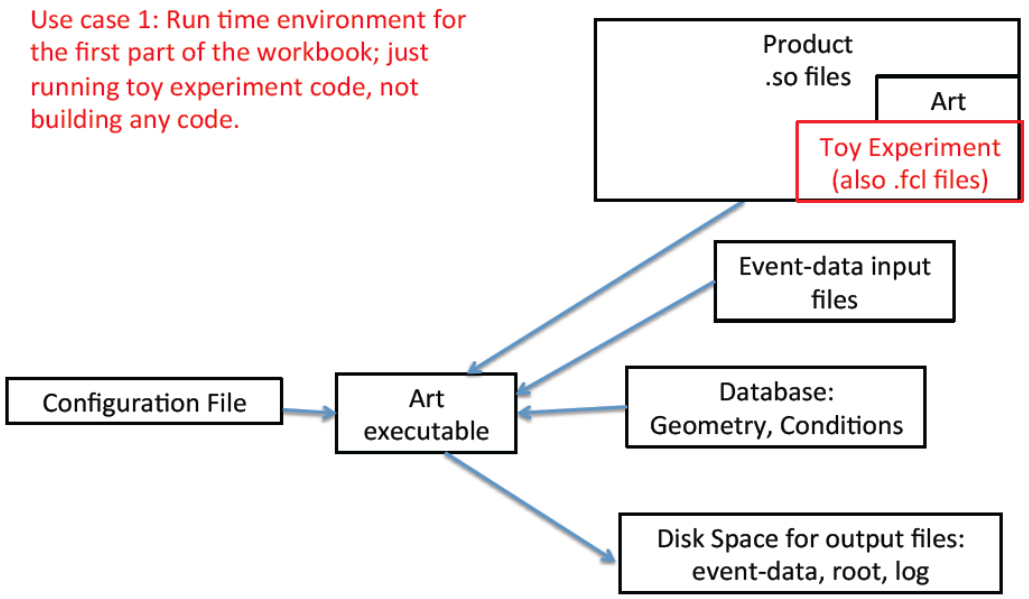


Figure 9.1: Elements of the *art* run-time environment for the first Workbook exercise

experiment, these are usually stored in a database that stands apart from the other elements in the picture.

The arrows in Figure 9.1 show the direction in which information flows. Everything but the output flows into the *art* executable.

9.5 The Input and Configuration Files for the Workbook Exercises

Several event-data input files have been provided for use by the Workbook exercises. These input files are packaged as part of the toyExperiment UPS product. Table 9.1 lists the range of event IDs found in each file. You will need to refer back to this table as you proceed.

A run-time configuration (FHiCL) file has been provided for each exercise. For Exercise 1 it is `hello.fcl`.

Table 9.1: Input files provided for the Workbook exercises

File Name	Run	SubRun	Range of Event Numbers
input01_data.root	1	0	1 . . . 10
input02_data.root	2	0	1 . . . 10
input03_data.root	3	0	1 . . . 5
	3	1	1 . . . 5
	3	2	1 . . . 5
input04_data.root	4	0	1 . . . 1000

9.6 Setting up to Run Exercise 1

9.6.1 Log In and Set Up

The intent of this section is for the reader to start from "zero" and execute an *art* job, without necessarily understanding each step, just to get familiar with the process. A detailed discussion of what these steps do will follow in Section 9.9.

Some steps are written as statements, others as commands to issue at the prompt. Notice that *art* takes the argument -c hello.fcl; this points *art* to the run-time configuration file that will tell it what to do and where to find the "pieces" on which to operate.

Most readers: Follow the steps in Section 9.6.1.1, then proceed directly to Section 9.7.

If you wish to manage your working directory yourself, skip Section 9.6.1.1, follow the steps in Section 9.6.1.2, then proceed to Section 9.7.

If you log out and wish to log back in to continue this exercise, follow the procedure outlined in Section 10.5.

9.6.1.1 Initial Setup Procedure using Standard Directory

1. Log in to the computer you chose in Section 8.3.

2. Follow the site-specific setup procedure; see Table 5.1.

3. Make the standard working directory then cd to it; substitute your kerberos principal for the string <username>. These commands, shown on two lines, can each be typed on a single line.

```
mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/\
  workbook-tutorial/pre-built
```

```
cd $ART_WORKBOOK_WORKING_BASE/<username>/\
  workbook-tutorial/pre-built
```

4. Setup the toyExperiment UPS product:

```
setup toyExperiment v0_00_15 -q$ART_WORKBOOK_QUAL:prof
```

5. Copy the scripts into your working directory:

```
cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .
```

6. Use the provided script to create the symbolic links needed by the FHiCL files:

```
source makeLinks.sh
```

7. See what you have in the directory:

```
ls
helloExample.log inputFiles makeLinks.sh
skipEvents.fcl
hello.fcl inputs.txt output writeFile.fcl
```

Proceed to Section 9.7.

9.6.1.2 Initial Setup Procedure allowing Self-managed Working Directory

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Table 5.1
3. Make a working directory and cd to it.
4. Setup the toyExperiment UPS product:

   ```
   setup toyExperiment v0_00_15 -q$ART_WORKBOOK_QUAL:prof
   ```

5. Copy the scripts into your working directory:

   ```
   cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .
   ```

6. Make a subdirectory named `output`. If you prefer, you can make this on some other disk and put a symbolic link to it from the current working directory; name the link `output`.
7. Create a symbolic link to allow the FHiCL files to find the input files:

   ```
   ln -s $TOYEXPERIMENT_DIR/inputFiles .
   ```

8. See what you have in the directory:

   ```
   ls
   helloExample.log inputFiles makeLinks.sh
   skipEvents.fcl
   hello.fcl inputs.txt output writeFile.fcl
   ```

Proceed to Section 9.7.

9.6.1.3 Setup for Subsequent Exercise 1 Login Sessions

If you log out and later wish to log in again to work on this exercise, you need to do the folllowing:

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Section 5.
3. cd to your working directory, e.g., for the standard case (shown here on two lines):

```
cd $ART_WORKBOOK_WORKING_BASE/<username>/\
  workbook-tutorial/pre-built
```

4. Setup the toyExperiment UPS product:

```
setup toyExperiment v0_00_15 -q$ART_WORKBOOK_QUAL:prof
```

Compare this with the list given in Section 9.6.1. You will see that three steps are missing because they only need to be done the first time.

You are now ready to run *art* as you were before.

9.7 Execute *art* and Examine Output

From your working directory, execute *art* on the FHiCL file `hello.fcl` and send the output to `output/hello.log`:

```
art -c hello.fcl >& output/hello.log
```

Compare the ouptut you produced (in the file `output/hello.log`) against Listing 9.1; the only differences should be the timestamps and some line breaking. *art* will have processed the first file listed in Table 9.1.

Listing 9.1: Sample output from running hello.fcl

```
1
2  %MSG-i MF_INIT_OK:  art 27-Apr-2013 21:22:13 CDT JobSetup
3  Messagelogger initialization complete.
4  %MSG
5  27-Apr-2013 21:22:14 CDT  Initiating request to open file
6  inputFiles/input01_data.root
7  27-Apr-2013 21:22:14 CDT  Successfully opened file
8  inputFiles/input01_data.root
9  Begin processing the 1st record. run: 1 subRun: 0 event: 1 at
10 27-Apr-2013 21:22:14 CDT
11 Hello World! This event has the id: run: 1 subRun: 0 event: 1
12 Begin processing the 2nd record. run: 1 subRun: 0 event: 2 at
13 27-Apr-2013 21:22:14 CDT
14 Hello World! This event has the id: run: 1 subRun: 0 event: 2
15 Hello World! This event has the id: run: 1 subRun: 0 event: 3
16 Hello World! This event has the id: run: 1 subRun: 0 event: 4
17 Hello World! This event has the id: run: 1 subRun: 0 event: 5
18 Hello World! This event has the id: run: 1 subRun: 0 event: 6
19 Hello World! This event has the id: run: 1 subRun: 0 event: 7
20 Hello World! This event has the id: run: 1 subRun: 0 event: 8
21 Hello World! This event has the id: run: 1 subRun: 0 event: 9
22 Hello World! This event has the id: run: 1 subRun: 0 event: 10
23 27-Apr-2013 21:22:14 CDT  Closed file inputFiles/input01_data.root
24
25 TrigReport ---------- Event  Summary ------------
26 TrigReport Events total = 10 passed = 10 failed = 0
27
28 TrigReport ------ Modules in End-Path: e1 ------------
29 TrigReport  Trig Bit#    Visited     Passed     Failed      Error Name
30 TrigReport     0    0         10         10          0          0 hi
31
32 TimeReport ---------- Time  Summary ---[sec]----
33 TimeReport CPU = 0.004000 Real = 0.002411
34
35 Art has completed and will exit with status 0.
```

Every time you run *art*, the first thing to check is the last line in your output or log file. It should be `Art has completed and will exit with status 0.` If the status is not 0, or if this line is missing, it is an error; please contact the *art* team as described in Section 3.4.

A future version of these instructions will specify how much disk space is needed, including space for all ouptut files.

9.8 Understanding the Configuration

The file `hello.fcl` gives *art* its run-time configuration. This file is written in the Fermilab Hierarchical Configuration Language (FHiCL, pronounced "fickle"), a language that was developed at Fermilab to support run-time configuration for several projects, including *art*. By convention, files written in FHiCL end in `.fcl`. As you work through the Workbook, the features of FHiCL that are relevant for each exericse will be explained.

art accepts some command line options that can be used in place of items in the FHiCL file. You will encounter some of these in this section.

The full details of the FHiCL language, plus the details of how it is used by *art*, are given in the Users Guide, Chapter 22. Most people will find it much easier to follow the discussion in the Workbook documentation than to digest the full documentation up front.

9.8.1 Some Bookkeeping Syntax

In a FHiCL file, the start of a comment is marked either by the hash sign character (#) or by a C++ style double slash (//); a comment may begin in any column.

The hash sign has one other use. If the first eight characters of a line are exactly `#include`, followed by whitespace and a quoted file path, then the line will be interpreted as an *include directive* and the line containing it will be replaced by the contents of the file named in the include directive.

The basic element of FHiCL is the *definition*, which has the form

```
name : value
```

A group of FHiCL definitions delimited by braces {} is called a *table*(γ). Within *art*, a FHiCL table gets turned into a C++ object called a *parameter set*(γ); this document set will often refer to a FHiCL table as a parameter set.

The fragment of `hello.fcl` shown below contains the FHiCL table that configures the *source*(γ) of events that *art* will read in and operate on.

```
source : {
  module_type : RootInput
  fileNames : [
  "inputfiles/input01_data.root"
  ]
}
```

At the outermost scope of the FHiCL file, *art* will interpret the `source` parameter set as the description of the source of events for this run of *art*.

`module_type` is an identifier that tells *art* the name of a module to load and run, `RootInput` in this case. `RootInput`, a standard source module provided by *art*, reads disk files containing event-data written in *art*-defined ROOT-based format.

The string `fileNames` defined in the `RootInput` module scope gives the module a list of filenames from which to read events.

The name `source` is an *identifier* in *art*; i.e., the name `source` has no special meaning to FHiCL but it does have a special meaning to *art*. To be precise, it only has a special meaning to *art* if it is at the outermost *scope*(γ) of a FHiCL file; i.e., not inside any braces {} within the file. The notion of *scope* in FHiCL is discussed further in Chapter 13. When *art* sees a parameter set named `source` at the outermost scope, then *art* will interpret that parameter set to be the description of the source of events for this run of *art*.

In the `source` parameter set, `module_type` is an identifier in *art* that tells *art* the name of a module that it should load and run, `RootInput` in this case. `RootInput` is one of the standard source modules provided by *art* and it reads disk files containing event-data written in an *art*-defined ROOT-based format. The default behavior of the `RootInput` module is to start at the first event in the first file and read to the end of the last event in the last file.*

The string `fileNames` is again an identifier, but this time defined in the `RootInput` module, that gives the module a list of filenames from which to read events. The list is delimited by square brackets and contains a comma-separated list of filenames. This exam-

* In the Workbook, the only source `module_type` that you will see will be `RootInput`. Your experiment may have a source module that reads events from the live experiment and other source modules that read files written in experiment-defined formats.

ple shows only one filename, but the square brackets are still required. The proper FHiCL name for a comma-separated list delimited by square brackets is a *sequence*(γ).

In most cases the filenames in the sequence must be enclosed in quotes. FHiCL, like many other languages has the following rule: if a string contains white space or any special characters, then quoting it is required, otherwise quotes are optional.

FHiCL has its own set of special characters; these include anything *except* all upper and lower case letters, the numbers 0 through 9 and the underscore character. *art* restricts the use of the underscore character in some circumstances; these will be discussed as they arise.

It is implied in the foregoing discussion that a FHiCL *value* need not be a simple thing, such as a number or a quoted string. For example, in `hello.fcl`, the value of `source` is a parameter set (of two parameters) and the value of `fileNames` is a (single-item) sequence.

9.8.2 Some Physics Processing Syntax

The identifier *physics*(γ), when found at the outermost scope, is an identifier reserved to *art*. The `physics` parameter set is so named because it contains most of the information needed to describe the physics workflow of an *art* job.

The fragment of `hello.fcl` below shows a rather long-winded way of telling *art* to find a module named `HelloWorld` and execute it.

```
physics : {

 analyzers : {
    hi: {
    module_type : HelloWorld
    }
 }

 e1 : [ hi ]
 end_paths : [e1 ]

}
```

At the outermost scope of the FHiCL file, *art* will interpret the `physics` parameter set as the description of the physics workflow for this run of *art*.

As a top-level identifier within the `physics` scope, `analyzers` defines for *art* the run-time configuration for all the analyzer modules in the job, e.g., `HelloWorld`.

`e1` is called a path; it is a FHiCL sequence of module labels; it is an arbitary identifier. `end_paths` is a FHiCL sequence of path names; it is an identifier reserved to *art*. Together they specify the workflow; see Section 9.8.8.

Why so long-winded? *art* has very powerful features that enable execution of multiple complex chains of modules; the price is that specifying something simple takes a lot of keystrokes.

Within the `physics` parameter set, notice the identifier `analyzers`. When found as a top-level identifier within the `physics` scope, it is recognized as a keyword reserved to *art*. The `analyzers` parameter set defines the run-time configuration for all of the analyzer modules that are part of the job – only `HelloWorld` in this case.

For our current purposes, the module `HelloWorld` does only one thing of interest, namely for every event it prints one line (shown here as three):

```
Hello World! This event has the id: run: <RR>
                                    subRun: <SS>
                                    event: <EE>
```

where RR, SS and EE are substituted with the actual run, subRun and event number of each event.

If you look back at Listing 9.1, you will see that this line appears ten times, once each for events 1 through 10 of run 1, subRun 0 (as expected, according to Table 9.1). The remainder of the listing is standard output generated by *art*.

The remainder of the lines in `hello.fcl` appears below. The line starting with *process_name*(γ) tells *art* that this job has a name and that the name is "hello"; it has no real significance in these simple exericses. However the name of the process must not contain any underscore characters; the reason for this restriction we be explained in Section 15.4.2.

The `services` parameter set provides run-time configuration information for all *art* services.

```
#include "fcl/minimalMessageService.fcl"

process_name : hello

services : {
  message : @local::default_message }
```

The identifier `process_name` tells *art* that the name of this job is `hello`.

The `services` parameter set provides the run-time configuration for all the required *art* services, in this case the message service. Its configuration is set in the file indicated in the #include directive.

For our present purposes, it is sufficient to know that the configuration for the message service is found inside the file that is included via the `#include` line. The message service controls the limiting and routing of debug, informational, warning and error messages generated by *art* or by user code. The message service does not control information written directly to `std::cout` or `std::cerr`.

9.8.3 *art* Command line Options

art supports some command line options. To see what they are, type the following command at the bash prompt:

```
art --help
```

Note that some options have both a short form and a long form. This is a common convention for Unix programs; the short form is convenient for interacive use and the long form makes scripts more readable. It is also a common convention that the short form of

an option begins single dash character, while the long form of an option begins with two dash characters, for example --help above.

9.8.4 Maximum Number of Events to Process

By default *art* will read all events from all of the specified input files. You can set a maximum number of events in two ways, one way is from the command line:

```
art -c hello.fcl -n 5 >& output/hello-n5.log
```

```
art -c hello.fcl --nevts 4 >& output/hello-nevts4.log
```

Run each of these commands and observe their output.

The second way is within the FHiCL file. Start by making a copy of `hello.fcl`:

```
cp hello.fcl hi.fcl
```

Edit `hi.fcl` and add the following line anywhere in the source parameter set:

```
maxEvents    : 3
```

By convention this is added after the fileNames definition but it can go anywhere inside the source parameter set because the order of parameters within a FHiCL table is not important. Run *art* again, using `hi.fcl`:

```
art -c hi.fcl >& output/hi.log
```

You should see output from the `HelloWorld` module for only the first three events.

To configure the file for *art* to process all the events, i.e., to run until *art* reaches the end of the input files, either leave off the maxEvents parameter or give it a value of -1.

If the maximum number of events is specified both on the command line and in the FHiCL file, then the command line takes precedence. Compare the outputs of the following commands:

```
art -c hi.fcl >& output/hi2.log
```

```
art -c hi.fcl -n 5 >& output/hi-n5.log
```

```
art -c hi.fcl -n -1 >& output/hi-neg1.log
```

9.8.5 Changing the Input Files

For historical reasons, there are multiple ways to specify the input event-data file (or the list of input files) to an *art* job:

- within the FHiCL file's `source` parameter set
- on the *art* command line via the -s option (you may specify one input file only)
- on the *art* command line via the -S option (you may specify a text file that lists multiple input files)
- on the *art* command line, after the last recognized option (you may specify one or more input files)

If input file names are provided both in the FHiCL file and on the command line, the command line takes precedence.

Let's run a few examples.

We'll start with the -s command line option (second bullet). Run *art* without it (again), for comparison (or recall its output from Listing 9.1):

```
art -c hello.fcl >& output/hello.log
```

To see what you should expect given the following input file, check Table 9.1, then run:

```
art -c hello.fcl -s inputFiles/input02_data.root >& output/hello-s.log
```

Notice that the 10 events in this output are from run 2 subRun 0, in contrast to the previous printout which showed events from run 1. Notice also that the command line specification overrode that in the FHiCL file. The -s (lower case) command line syntax will only permit you to specify a single filename.

This time, edit the source parameter set inside the `hi.fcl` file (first bullet); change it to:

```
source : {
  module_type : RootInput
  fileNames   : [ "inputFiles/input01_data.root",
                  "inputFiles/input02_data.root" ]
  maxEvents   : -1
}
```

(Notice that you also added `maxEvents : -1`.) The names of the two input files could have been written on a single line but this example shows that, in FHiCL, newlines are treated simply as white space.

Check Table 9.1 to see what you should expect, then rerun *art* as follows:

```
art -c hi.fcl >& output/hi-2nd.log
```

You will see 20 lines from the `HelloWorld` module; you will also see messages from *art* at the open and close operations on each input file.

Back to the -s command-line option, run:

```
art -c hi.fcl -s inputFiles/input03_data.root >& output/run3.log
```

This will read only `inputFiles/input03_data.root` and will ignore the two files specified in the `hi.fcl`. The output from the `HelloWorld` module will be the 15 events from the three subRuns of run 3.

There are several ways to specify multiple files at the command line. One choice is to use the -S (upper case) [--source-list] command line option (third bullet) which takes as its argument the name of a text file containing the filename(s) of the input event-data file(s). An example of such as file has been provided, `inputs.txt`. Look at the contents of this file:

```
cat inputs.txt
```

```
inputFiles/input01_data.root
inputFiles/input02_data.root
inputFiles/input03_data.root
```

Now run *art* using `inputs.txt` to specify the input files:

```
art -c hi.fcl -S inputs.txt >& output/file010203.log
```

You should see the `HelloWorld` output from the 35 events in the three files; you should also see the messages from *art* about the opening and closing of the three files.

Finally, you can list the input files at the end of the *art* command line (fourth bullet):

```
art -c hi.fcl inputFiles/input02_data.root inputFiles/input03_data.root >& output/file0203.log
```

(Remember the Unix convention about a trailing backslash marking a command that continues on another line; see Chapter 2.) In this case you should see the `HelloWorld` output from the 25 events in the two files.

In summary, there are three ways to specify input files from the command line; all of them override any input files specified in the FHiCL file. Do not try to use two or more of these methods on a single *art* command line; the *art* job will run without issuing any messages but the output will likely be different than you expect.

9.8.6 Skipping Events

The `source` parameter set supports a syntax to start execution at a given event number or to skip a given number of events at the start of the job. Look, for example, at the file `skipEvents.fcl`, which differs from `hello.fcl` by the addition of two lines to the source parameter set:

```
firstEvent   : 5
maxEvents    : 3
```

art will process events 5, 6, and 7 of run 1, subRun 0. Try it:

```
art -c skipEvents.fcl >& output/skipevents1.log
```

An equivalent operation can be done from the command line in two different ways. Try the following two commands and compare the output:

```
art -c hello.fcl -e 5 -n 3 >& output/skipevents2.log
```

```
art -c hello.fcl --nskip 4 -n 3 >& output/skipevents3.log
```

You can also specify the intial event to process relative to a given event ID (which, recall, contains the run, subRun and event number). Edit `hi.fcl` and edit the `source` parameter set as follows:

```
source : {
  module_type : RootInput
  fileNames   : [ "inputFiles/input03_data.root" ]
  firstRun    : 3
  firstSubRun : 1
  firstEvent  : 6
}
```

When you run this job, *art* will process events starting from run 3, subRun 2, event 1, – because there are only 5 events in subRun 1.

```
art -c hi.fcl >& output/startatrun3.log
```

9.8.7 Identifying the User Code to Execute

Recall from Section 9.8.2 that the `physics` parameter set contains the physics content for the *art* job. Within this parameter set, *art* must be able to determine which (user code) modules to process. These must be referenced via *module labels*(γ), which as you will see, represent the pairing of a module name and a run-time configuration.

Look back at the listing on page 126, which contains the `physics` parameter set from `hello.fcl`. The `analyzer` parameter set, nested inside the `physics` parameter set, contains the definition:

```
hi : {
  module_type : HelloWorld
}
```

The identifier `hi` is a module label (defined by the user, not by FHiCL or *art*) whose value must be a parameter set that *art* will use to configure a module. The parameter set for a module label must contain (at least) a FHiCL definition of the form:

```
module_type : <module-name>
```

Here `module_type` is an identifier reserved to *art* and `<module-name>` tells *art* the name of the module to load and execute. (Since it is within the `analyzer` parameter set, the module must be of type EDAnalyzer; i.e. the *base type* of `<module-name>` must be EDAnalyzer.)

Module labels are fully described in Section 22.5.

In this example *art* will look for a module named `HelloWorld`, which it will find as part of the toyExperiment UPS product. Section 9.10 describes how *art* uses `<module-name>` to find the dynamic library that contains code for the `HelloWorld` module. A parameter set that is used to configure a module may contain additional lines; if present, the meaning of those lines is understood by the module itself; those lines have no meaning either to *art* or to FHiCL.

Now look at the FHiCL fragment below that starts with `analyzers`. We will use it to reinforce some of the ideas discussed in the previous paragraph.

art allows you to write a FHiCL file that uses a given module more than once. For example you may want to run an analysis twice, once with a loose mass cut on some intermediate state and once with a tight mass cut on the same intermediate state. In *art* you can do this by writing one module and making the cuts "run-time configurable." This idea will be developed further in Chapter 14.

```
analyzers : {
```

When *art* processes this fragment it will look for a module named `MyAnalysis` and instantiate it twice...

```
    loose: {
    module_type : MyAnalysis
    mass_cut : 20.
    }
```

... once using the parameter set labeled `loose` ...

```
    tight: {
    module_type : MyAnalysis
    mass_cut : 15.
    }
```

... and once using the parameter set labeled `tight`.

```
}
```

The two instances of the module `MyAnalysis` are distinguished by the module labels `tight` and `loose`. *art* requires that module labels be unique within a FHiCL file. Module labels may contain only upper- and lower-case letters and the numerals 0 to 9.

In the FHiCL files in this exercise, all of the modules are analyzer modules. Since analyzers do not make data products, these module labels are nothing more than identifiers inside the FHiCL file. For producer modules, however, which *do* make data products, the module label becomes part of the data product identifier and as such has a real signficance. All module labels must conform to the same naming rules.

Within *art* there is no notion of reserved names or special names for module labels; however your experiment will almost certainly have established some naming conventions.

9.8.8 Paths and the *art* Workflow

In the `physics` parameter set in `hello.fcl` there are two parameter definitions that, taken together, specify the *workflow* of the *art* job:

```
physics {
  e1         : [ hi ]
  end_paths  : [ e1 ]
}
```

where *workflow* means which modules *art* should run and in which order.[†]

In this exercise there is only one module to run so the workflow is trivial: for each event, run the module with the label `hi`. As you work through the Workbook you will encounter workflows that are more complex and they will be described as you encounter them.

The FHiCL parameter `e1` is called a *path*. A path is simply a FHiCL sequence of module labels. The name of a path can be any user-defined name that satisfies the following:

1. It must be defined as part of the `physics` parameter set.
2. It must be a valid FHiCL name.

[†] The word *workflow* is used widely in the computing world. When a complete job comprises several discrete tasks, a workflow specifies the order in which the tasks should be performed. The full story of workflows is very rich and this sub-section will give only cover the parts needed to understand Workbook Exercise 1.

3. It must be unqiue within the *art* job.

4. It must NOT be one of the following 5 names that are reserved to *art*: `analyzers`, `filters`, `producers`, `end_paths` and `trigger_paths`.

Another way of saying the first item is that it must be defined "at physics scope".

An *art* job may contain many paths, each of which is a FHiCL sequence of module labels. When many groups are working on a common project, this helps to maximize the independence of each work group.

The parameter `end_paths` is not itself a path! Rather it is a FHiCL sequence of path names. It is also a name that is reserved to *art*. When *art* wants to learn the workflow that it should execute it starts by looking at the `end_paths` parameter. Using this information it can find all of the module labels that are part of the workflow.

If a path is listed in the `end_paths` parameter, then it may contain module labels only for analyzer and output modules. A similar mechansim is used to specify the workflow of producer and filter modules; that mechanism will be discussed when you encounter it. If you need a reminder about the types of modules, see Section 3.6.3.

If the `end_paths` parameter is absent or defined as:

```
end_paths : [ ]
```

art will understand that this job has no analyzer modules and no output modules to execute. It is legal to define a path as an empty FHiCL sequence.

As is standard in FHiCL, if the definition of `end_paths` appears more than once, the last definition takes precendence.

The notion of path introduced in this section is the third thing in the *art* documentation suite that is called a path. The other two are the notion of a path in a filesystem and the notion of an environment variable that is a colon-delimited set of directory names. The use should be clear from the context; if it is not, please let the authors of the workbook know; see Section 3.4.

If you find the above description unsatisfying or incomplete. the next section, 9.8.8.1, has a more detail. You can choose to read it now or come back to it later.

9.8.8.1 Paths and the *art* Workflow: Details

This material is optional; it contains more details about the material just described in Section 9.8.8. It is not really a dangerous bend section — just a side trip.

Exercise 1 is not rich enough to illustrate how to specify an *art* workflow so let's construct a richer example.

Suppose that there are two groups of people working on a large collaborative project, the project leaders are Anne and Rob. Each group has a workflow that requires running 5 or 6 module instances; some of the module instances may be in the workflow for both groups. Recall that an instance of a module refers to the name of a module plus its parameter set; a module instance is specified by giving its module label. For this example let's have 8 module instances with the unimaginative names, `a` through `h`. The workflow for this example might look something like:

```
anne      : [ a, b, c, d, e, h]
rob       : [ a, b, f, c, g ]
end_paths : [ anne, rob ]
```

That is, Anne defines the modules that her group needs to run and Rob defines the modules that his group needs to run. Anne does not need to know anything about Rob's list and Rob does not need to know anything about Anne's list. The parameter definitions `anne` and `rob` are called *paths*; each is a list of module labels. The rules for legal path names were given in Section 9.8.8.

The parameter named `end_paths` is not itself a path, rather it is a FHiCL sequence of paths. Moreover it has a special meaning to *art*. During *art*'s initialization phase, it needs to learn the workflow for the job. The first step is to find the parameter named `end_paths`, defined within the `physics` parameter set. When *art* processes the definition of `end_paths` it will form the set of all module labels found in the contributing paths, with any duplicates removed. For this example, the list might look something like: `[a, b, c, d, e, h, f, g]` . When *art* processes an event, this is the set of module instances that it will execute. The order in which the module instances are executed is discussed in Section 9.8.8.2.

The above machinery probably seems a little heavyweight for the example given. But consider a workflow like that needed to design the trigger for the CMS experiment: it

had a about 200 paths and many hundreds of modules. Finding the set of unique modules labels is not a task that you would want to do by hand! By introducing the idea of paths, the design allows each group can focus on it own work, without needing ongoing, detailed consultation with every other group.

Actually, the above story is only half of the story: the module labels given in the paths `anne` and `rob` may only be the labels of analyzer or output modules. There is a parallel mechanism to specify the workflow for producer and filter modules.

To illustrate this parallel mechanism let's continue the above example of two work groups lead by Rob and Anne. In this case let there be filter modules with labels given by, `f0`, `f1`, `f2` ... and producer modules with labels given by `p0`, `p1`, `p2` .... In this example, a workflow might look something like:

```
t_anne         : [ p0, p1, p2, f0, p3, f1 ]
t_rob          : [ p0, p1, f2, p2, f0, p4 ]
trigger_paths  : [ t_anne, t_rob ]

e_anne         : [ a, b, c, d, e ]
e_rob          : [ a, b, f, c, g ]
end_paths      : [ e_anne, e_rob ]
```

Here the parameters `t_anne`, `e_anne`, `t_rob`, and `e_rob` are all the names of paths. All must be be legal FHiCL parameter names, be unique within an *art* job and not conflict with identifiers reserved to *art* at physics scope. In this example the path names are prefixed with `t_` for paths that will be put into the `trigger_paths` parameter and with `e_` for paths that will be put into the `end_paths` parameter. This is just to make it easier for you to follow the example; the prefixes have no intrinsic meaning.

During *art*'s initialization phase it processes `trigger_paths` in the same way that it processed `end_paths`: it forms the set of all module labels found in the contributing paths, with duplicates removed. The order in which the module instances are executed is discussed in Section 9.8.8.2.

What happens if you define a path with a mix of modules from the two groups:

```
bad_path       : [ p0, p1, p2, f0, p3, f1, a, b ]
end_paths      : [ e_anne, e_rob, bad_path ]
```

In this case *art* (not FHiCL) will recognize that there are producer and filter modules specified in a path that contributes to `end_paths`; *art* will then print a diagnostic message and stop. This will occur very early in *art*'s initialization phase so you will get reasonably prompt feedback. Similarly, if *art* discovers that there are analyer or output modules in any of the paths contributing to `trigger_paths`, it will then print a diagnostic message and stop.

If you put a module label into either `end_paths` or `trigger_paths`, *art* will print a diagnostic message and stop. If you put a path name into the definition of another path *art* will print a diagnostic message and stop.

Now it's time to define two really badly chosen names:[‡] *trigger paths* and *end paths*, both without underscores! In the above fragment the paths prefixed with `t_` are called *trigger paths*, without an underscore; they are so named because they contain module labels for only producer and filter modules; therefore they are paths that satisfy the rules for inclusion in the definition of `trigger_paths` parameter. Similarly the paths prefixed with `e_` are called *end paths* because they satisfy the rules for inclusion in the definition of `end_paths` parameter.

While this documentation will try to avoid avoid confusion between *end paths* and `end_paths` you will certainly encounter careless use of the two ideas when you read other documentation or discuss your FHiCL files with your colleagues. There should always be enough context to sort out the meaning but it might take work.

9.8.8.2 Order of Module Execution

If the `trigger_paths` parameter contains a single *trigger path*, then *art* will execute the modules in that *trigger path* in the order that they are specified.

When more than one *trigger path* is present in `trigger_paths`, *art* will choose one of the *trigger path*s and execute its module instances in order. It will then choose a second *trigger path*. If any module instances in this path were already executed in the first *trigger path*, *art* will not execute them a second time; it will execute the remaining module instances in the order specified by the second *trigger path*. And so on for any remaining *trigger path*s.

[‡] It could have been worse. They could have been named on Opposite Day!

The rules for order of execution of module instances named in an *end path* are different. Since analyzer and output modules may neither add new information to the event nor communicate with each other except via the event, the processing order is not important. By definition, then, *art* may run analyzer and output modules in any order. In a simple *art* job with a single path, *art* will, in fact, run the modules in the order of appearance in the path, but do not write code that depends on execution order because *art* is free to change it.

9.8.9 Writing an Output File

The file `writeFile.fcl` gives an example of writing an output file. Open the file in an editor and find the parts of the file that are discussed below.

Output files are written by output modules; one module can write one file. An *art* job may run zero or more output modules.

If you wish to add an output module to an *art* job there three steps:

1. Create a parameter set named `outputs` at the outermost scope of the FHiCL file. The name `outputs` is prescribed by *art*.
2. Inside the `outputs` parameter set, add a parameter set to configure an output module. In `writeFile.fcl` this parameter set has the module label `output1`.
3. Add the module label of the output module to an *end path* (not to the `end_paths` parameter but to one of the paths that is included in `end_paths`). In `writeFile.fcl` the module label `output1` is added to the *end path* `e1`.

If you wish to write more output files, repeat steps 2 and 3 for each additional output file.

The parameter set `output1` tells *art* to make a module whose type is `RootOutput`. The class `RootOutput` is a standard module that is part of *art* and that writes events from memory to a disk file in an *art*-defined, ROOT-based format. The `fileName` parameter specifies the name of the output file; this parameter is processed by the `RootOutput` module. Files written by the module `RootOutput` can be read by the module `RootInput`. The identifier `output1` is just another module label that obeys the rules discussed in Section 9.8.7.

In the example of `writeFile.fcl` the output module takes its default behaviour: it will write all of the information about each input event to the output file. `RootOutput` can be configured to:

1. write only selected events
2. for each event write only a subset of the available data products.

How to do this will be described in section that will be written later.

Before running the exercise, look at the source parameter set of `writeFile.fcl`; note that it is configured to read only events 4, 5, 6, and 7.

To run writeFile.fcl and check that it worked correctly:

```
art -c writeFile.fcl
```

```
ls -s output/writeFile_data.root
```

```
art -c hello.fcl -s output/writeFile_data.root
```

The first command will write the ouptut file; the second will check that the output file was created and will tell you its size; the last one will read back the output file and print the event IDs for all of the events in the file. You should see the `HelloWorld` printout for events 4, 5, 6 and 7.

9.9 Understanding the Process for Exercise 1

Section 9.6.1 contained a list of steps needed to run this exercise; this section will describe each of those steps in detail. When you understand what is done in these steps, you will understand the run-time environment in which *art* runs. As a reminder, the steps are listed again here. The commands that span two lines can be typed on a single line.

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Chapter 5
3. `mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/\`

```
  workbook-tutorial/pre-built
```

In the above and elsewhere as indicated, substitute your kerberos principal for the string <username>.

4. `cd $ART_WORKBOOK_WORKING_BASE/<username>/\`
   `  workbook-tutorial/pre-built`
5. `setup toyExperiment v0_00_15 -q$ART_WORKBOOK_QUAL:prof`
6. `cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .`
7. `source makeLinks.sh`
8. Run *art*:

   ```
   art -c hello.fcl >& output/hello.log
   ```

Steps 1 and 4 should be self explanatory and will not be discussed further.

When reading this section, you do not need to run any of the commands given here; this is a commentary on commands that you have already run.

9.9.1 Follow the Site-Specific Setup Procedure (Details)

The site-specific setup procedure, described in Chapter 5, ensures that the UPS system is properly initialized and that the UPS database (containing all of the UPS products needed to run the Workbook exercises) is present in the PRODUCTS environment variable.

This procedure also defines two environment variables that are defined by your experiment to allow you to run the Workbook exercises on their computer(s):

ART_WORKBOOK_WORKING_BASE the top-level directory in which users create their working directory for the Workbook exercises

ART_WORKBOOK_OUTPUT_BASE the top-level directory in which users create their output directory for the Workbook exercises; this is used by the script `makeLinks.sh`

If these environment variables are not defined, ask a system admin on your experiment.

9.9.2 Make a Working Directory (Details)

On the Fermilab computers the home disk areas are quite small so most experiments ask that their collaborators work in some other disk space. This is common to sites in general, so we recommend working in a separate space as a best practice. The Workbook is designed to require it.

This step, shown on two lines as:

```
mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/workbook-tutorial/\
  pre-built
```

creates a new directory to use as your working directory. It is defined relative to an environment variable described in Section 9.9.1. It only needs to be done the first time that you log in to work on Workbook exercises.

If you follow the rest of the naming scheme, you will guarantee that you have no conflicts with other parts of the Workbook.

As discussed in Section 9.6.1.2, you may of course choose your own working directory on any disk that has adequate disk space.

9.9.3 Setup the toyExperiment UPS Product (Details)

This step is the main event in the eight-step process.

```
setup toyExperiment v0_00_14 -q$ART_WORKBOOK_QUAL:prof
```

This command tells UPS to find a product named toyExperiment, with the specified version and qualifiers, and to *setup* that product, as described in Section 7.3.

The required qualifiers may change from one experiment to another and even from one site to another within the same experiment. To deal with this, the site specific setup procedure

defines the environment variable ART_WORKBOOK_QUAL, whose value is the qualifier string that is correct for that site.

The complete UPS qualifier for toyExperiment has two components, separated by a colon: the string defined by ART_WORKBOOK_QUAL plus a qualifier describing the compiler optimization level with which the product was built, in this case "prof"; see Section 3.6.7 for information about the optimization levels.

Each version of the toyExperiment product knows that it requires a particular version and qualifier of the *art* product. In turn, *art* knows that it depends on particular versions of ROOT, CLHEP, boost and so on. When this recursive setup has completed, over 20 products will have been setup. All of these products define environment variables and about two-thirds of them add new elements to the environment variables PATH and LD_LIBRARY_PATH.

If you are interested, you can inspect your environment before and after doing this setup. To do this, log out and log in again. Before doing the setup, run the following commands:

```
printenv > env.before
```

```
printenv PATH | tr : \\n > path.before
```

```
printenv LD_LIBRARY_PATH | tr : \\n > ldpath.before
```

Then setup toyExperiment and capture the environment afterwards (env.after). Compare the before and after files: the after files will have many, many additions to the environment. (The fragment `| tr : \\n` tells the bash shell to take the output of printenv and replace every occurrence of the colon character with the newline character; this makes the output much easier to read.)

9.9.4 Copy Files to your Current Working Directory (Details)

The step:

```
cp $TOYEXPERIMENT_DIR/HelloWorldScripts/* .
```

only needs to be done only the first time that you log in to work on the Workbook.

In this step you copied the files that you will use for the exercises into your current working directory. You should see these files:

```
hello.fcl  makeLinks.sh  skipEvents.fcl  writeFile.fcl
```

9.9.5 Source makeLinks.sh (Details)

This step:

```
source makeLinks.sh
```

only needs to be done only the first time that you log in to work on the Workbook. It created some symbolic links that *art* will use.

The FHiCL files used in the Workbook exercises look for their input files in the subdirectory `inputFiles`. This script made a symbolic link, named inputFiles, that points to:

```
$TOYEXPERIMENT_DIR/inputFiles
```

in which the necessary input files are found.

This script also ensures that there is an output directory that you can write into when you run the exercises and adds a symbolic link from the current working directory to this output directory. The output directory is made under the directory $ART_WORKB0OK_OUTPUT_BASE; this environment variable was set by the site-specific setup procedure and it points to disk space that will have enough room to hold the output of the exercises.

9.9.6 Run *art* (Details)

Issuing the command:

```
art -c hello.fcl
```

runs the *art* main program, which is found in $ART_FQ_DIR/bin. This directory was added to your PATH when you setup toyExperiment. You can inspect your PATH to see that this directory is indeed there.

9.10 How does *art* find Modules?

When you ran `hello.fcl`, how did *art* find the module `HelloWorld`?

It looked at the environment variable LD_LIBRARY_PATH, which is a colon-delimited set of directory names defined when you setup the toyExperiments product. We saw the value of LD_LIBRARY_PATH in Section 9.9.3; to see it again, type the following:

```
printenv LD_LIBRARY_PATH | tr : \\n
```

The output should look similar to that shown in Listing 9.2.

Listing 9.2: Example of the value of LD_LIBRARY_PATH

```
1  /ds50/app/products/tbb/v4_1_2/Linux64bit+2.6-2.12-e2-prof/lib
2  /ds50/app/products/sqlite/v3_07_16_00/Linux64bit+2.6-2.12-prof/lib
3  /ds50/app/products/libsigcpp/v2_2_10/Linux64bit+2.6-2.12-e2-prof/lib
4  /ds50/app/products/cppunit/v1_12_1/Linux64bit+2.6-2.12-e2-prof/lib
5  /ds50/app/products/clhep/v2_1_3_1/Linux64bit+2.6-2.12-e2-prof/lib
6  /ds50/app/products/python/v2_7_3/Linux64bit+2.6-2.12-gcc47/lib
7  /ds50/app/products/libxml2/v2_8_0/Linux64bit+2.6-2.12-gcc47-prof/lib
8  /ds50/app/products/fftw/v3_3_2/Linux64bit+2.6-2.12-gcc47-prof/lib
9  /ds50/app/products/root/v5_34_05/Linux64bit+2.6-2.12-e2-prof/lib
10 /ds50/app/products/boost/v1_53_0/Linux64bit+2.6-2.12-e2-prof/lib
11 /ds50/app/products/cpp0x/v1_03_15/slf6.x86_64.e2.prof/lib
12 /ds50/app/products/cetlib/v1_03_15/slf6.x86_64.e2.prof/lib2
13 /ds50/app/products/fhiclcpp/v2_17_02/slf6.x86_64.e2.prof/lib
14 /ds50/app/products/messagefacility/v1_10_16/slf6.x86_64.e2.prof/lib
15 /ds50/app/products/art/v1_06_00/slf6.x86_64.e2.prof/lib
16 /ds50/app/products/toyExperiment/v0_00_14/slf6.x86_64.e2.prof/lib
17 /grid/fermiapp/products/common/prd/git/v1_8_0_1/Linux64bit-2/lib
```

Of course the leading element of each directory name, `/ds50/app` will be replaced by whatever is correct for your experiment. The last element in LD_LIBRARY_PATH is not relevant for running *art* and it may or may not be present on your machine, depending on details of what is done inside your site-specific setup procedure.

If you compare the names of the directories listed in LD_LIBRARY_PATH to the names of the directories listed in the PRODUCTS environment variable, you will see that all of these directories are part of the UPS products system. Moreover, for each product, the version, flavor and qualifiers are embedded in the directory name. In particular, both *art* and toyExperiment are found in the list.

If you ls the directories in LD_LIBRARY_PATH you will find that each directory contains many dynamic object library (`.so` files).

When *art* looks for a module named `HelloWorld`, it looks through the directories defined in
LD_LIBRARY_PATH and looks for a file whose name matches the pattern,

```
lib*HelloWorld_module.so
```

where the asterisk matches (zero or) any combination of characters. *art* finds that, in all of the directories, there is exactly one file that matches the pattern, and it is found in the directory (shown here on two lines):

```
/ds50/app/products/toyExperiment/v0_00_14/
     slf6.x86_64.e2.prof/lib/
```

The name of the file is:

```
libtoyExperiment_Analyzers_HelloWorld_module.so
```

If *art* had found no files that matched the pattern, it would have printed a diagnostic message and stopped execution. If *art* had found more than one file that matched the pattern, it would have printed a different diagnostic message and stopped execution. If this second error occurs it is possible to tell *art* which of the matches to choose; how to do this will be covered in a future chapter.

9.11 How does *art* find FHiCL Files?

This section will describe where *art* looks for FHiCL files. There are two cases: looking for the file specified by the command line argument -c and looking for files that have been included by a `#include` directive within a FHiCL file.

9.11.1 The -c command line argument

When you issued the command

```
art -c hello.fcl
```

art looked for a file named `hello.fcl` in the current working directory and found it. You may specify any absolute or relative path as the argument of the -c option. If *art* had not found `hello.fcl` in this directory it would have looked for it relative to the path defined by the environment variable FHICL_FILE_PATH. This is just another path-type environment variable, like PATH or LD_LIBRARY_PATH. You can inspect the value of FHICL_FILE_PATH by:

```
printenv FHICL_FILE_PATH
.:<some-directory-structure>products//toyExperiment/v0_00_15
```

In this case, the output will show the translated value of the environment variable TOY-EXPERIMENT_DIR. The presence of the current working directory (dot) in the path is redundant when processing the command line argument but it is significant in the case discussed in the next section.

Some experiments have chosen to configure their version of the *art* main program so that it will not look for the command line argument FHiCL file in FHICL_FILE_PATH. It is also possible to configure *art* so that only relative paths, not absolute paths, are legal values of the -c argument. This last option can be used to help ensure that only version-controlled files are used when running production jobs. Experiments may enable or disable either of these options when their main program is built.

9.11.2 `#include` Files

Section 9.8 discussed the listing on page 127, which contains the fragments of `hello.fcl` that are related to configuring the message service. The first line in that listing is an include directive. *art* will look for the file named by the include directive relative to FHICL_FILE_PATH and it will find it in:

```
$TOYEXPERIMENT_DIR/fcl/minimalMessageService.fcl
```

This is part of the toyExperiment UPS product.

The version of *art* used in the Workbook does not consider the argument of the include directive as an absolute path or as a path relative to the current working directory; it only looks for files relative to FHICL_FILE_PATH. This is in contrast to the choice made when processing the -c command line option.

When building *art*, one may configure *art* to first consider the argument of the include directive as a path and to consider FHICL_FILE_PATH only if that fails.

10 Exercise 2: Building and Running Your First Module

10.1 Introduction

In this exercise you will build and run a simple *art* module. Section 3.6.7 introduced the idea of a build system, a software package that compiles and links your source code to turn it into machine code that the computer can execute. In this chapter you will be introduced to the *art* development environment, which adds the following to the run-time environment (discussed in Section 9.4):

1. a build system
2. a source code repository
3. a working copy of the Workbook source code
4. a directory containing dynamic libraries created by the build system

In this and all subsequent Workbook exercises, you will use the build system used by the *art* development team, **cetbuildtools**. This system will require you to open two shell windows your local machine and, in each one, to log into the remote machine *. The windows will be referred to as the *source window* and the *build window*:

- In the *source window* you will check out and edit source code.
- In the *build window* you will build and run code.

*cetbuildtools requires what are called *out-of-source builds*; this means that the source code and the working space for the build system must be in separate directories.

Exercise 2 and all subsequent Workbook exercises will use the setup instructions found in Sections 10.4 and 10.5.

10.2 Prerequisites

Before running this exercise, you need to be familiar with the material in Part I (Introduction) of this documentation set and Chapter 9 from Part II (Workbook). Concepts that this chapter refers to include:

- namespace
- `#include` directives
- header file
- class
- constructor
- destructor
- the C preprocessor
- member function (aka method)
- const vs non-const member function
- argument list of a function
- signature of a function
- *declaration* vs *defintion* of a class
- arguments passed by reference
- arguments passed by const reference
- notion of *type*: e.g., a class, a struct, a free function or a typedef

In this chapter you will also encounter the C++ idea of *inheritance*. Understanding inheritance is not a prerequisite; it will be described as you encounter it in the Workbook exercises. Inheritance includes such ideas as,

- base class
- derived class
- virtual function
- pure virtual function
- concrete class

10.3 What You Will Learn

In this exercise you will learn:

- how to establish the *art* development environment
- how to checkout the Workbook exercises from the `git` source code management system
- how to use the **cetbuildtools** build system to build the code for the Workbook exercises
- how include files are found
- what a *link list* is
- where the build system finds the link list
- what the `art::Event` is and how to access it
- what the `art::EventID` is and how to access it
- what makes a class an *art module*
- where the build system puts the `.so` files that it makes

10.4 Initial Setup to Run Exercises: Standard Procedure

10.4.1 "Source Window" Setup

Up through step 3 of the procedure in this section, the results should look similar to those of Exercise 1. Note that the directory name chosen here in the mkdir step is different than that chosen in the first exercise; this is to avoid file name collisions.

If you want to use a self-managed working directory, in step 3 make a directory of your choosing and cd to it rather than to the directory shown.

In your source window do the following:

1. Log in to the computer you chose in Section 8.3.

2. Follow the site-specific setup procedure; see Table 5.1.

3. Make a new working directory and cd to it. Remember that you can type this first command, and all subsequent commands in the Workbook that are shown on two lines for formatting reasons, on a single line.

   ```
   mkdir -p $ART_WORKBOOK_WORKING_BASE/<username>/\
     workbook
   ```

   ```
   cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook
   ```

4. Set up the source code management system git and use it to pull down the workbook code to the directory `art-workbook`, which will be referred to as your *source* directory. The output for each step is explained in Section 10.4.2.1:

   (a) `setup git`

(b) `git clone http://cdcvs.fnal.gov/projects/art-workbook`

(c) `cd art-workbook`

(d) `git checkout -b v0_00_18 v0_00_18`

5. Source the script that sets up the environment properly:

```
source ups/setup_deps -p $ART_WORKBOOK_QUAL
```

The git commands are discussed in Section 10.4.2.1. The final step sources a script that defines a lot of environment variables — the same set that will be defined in the build window.

10.4.2 Examine Source Window Setup

10.4.2.1 About git and What it Did

Git is a source code management system[†] that is used to hold the source code for the Workbook exercises. A source code managment system is a tool that looks after the book-keeping of the development of a code base; among many other things it keeps a complete history of all changes and allows one to get a copy of the source code as it existed at any time in the past. Because of git's many advanced features, many HEP experiments are moving to git. git is fully described in the git manual[‡].

Some experiments set up git in their site-specific setup procedure; others do not. In running `setup git`, you have ensured that a working copy of git is in your PATH[§].

The `git clone` and `git checkout` commands produce a working copy of the Workbook source files in your source directory. Figure 10.1 shows a map of the source directory structure

[†]Other source code management systems with which you may be familar are CVS and SVN.

[‡]Several references for git can be found online; the "official" documentation is found at http://git-scm.com/documentation.

[§]No version needs to be supplied because the git UPS product has a current version declared; see Section 7.4.

created by the git commands. It does not show all the contents in each subdirectory. Note that the `.git` (hidden) directory under the source directory is colored differently; this is done to distinguish it from the rest of the contents of the source directory structure:

- When you ran git clone in Section 10.4.1, it copied the entire contents of the remote repository into this directory. The `.git` directory contains your local copy of the repository.
- When you ran git checkout, it created the rest of the structure under the source directory (what we call your "working area") and copied the requested version of everything you need from `.git` into this structure.

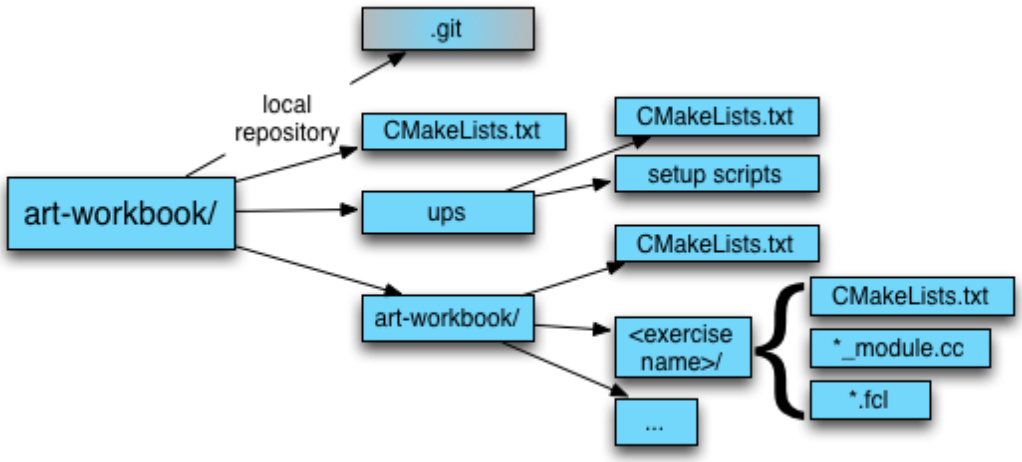


Figure 10.1: Representation of the reader's source directory structure (an `admin` directory is not shown)

; git clone should produce the following output:

```
Cloning into 'art-workbook'...
```

Executing the git checkout command should produce the following output:

```
Switched to a new branch 'v0_00_18 '
```

If you wish to learn about `git` branches, for the time being, you will need to consult a `git` manual.

If you do not see the expected output, contact the *art* team as described in Section 3.4.

10.4.2.2 Contents of the Source Directory

Figure 10.1 shows roughly what your source directory contains at the end of the setup procedure. You can see the correspondance between it and the output of the ls -a command:

```
cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/art-workbook
```

```
ls -a
```

```
. .. admin art-workbook CMakeLists.txt .git ups
```

Notice that it contains a subdirectory of the same name as its parent, `art-workbook`.

- The `admin` directory (not shown in Figure 10.1) contains some scripts used by **cetbuildtools** to customize the configuration. of the development environment.
- The `art-workbook` directory contains the main body of the source code for the Workbook exercises.
- The file `CMakeLists.txt` is the file that the build system reads to learn what steps it should do.
- The `ups` directory contains information about what UPS products this product depends on; it contains additional information used to configure the development environment.

Look inside the `art-workbook` (sub)directory (via ls) and see that it contains several files and subdirectories. The file `CMakeLists.txt` contains more instructions for the build system. Actually, every directory contains a `CMakeLists.txt`; each contains additional instructions for the build system. The subdirectory `FirstModule` contains the files that will be used in this exericse; the remaining subdirectories contain files that will be used in subsequent Workbook exercises.

If you look inside the `FirstModule` directory, you will see

```
CMakeLists.txt FirstAnswer01_module.cc First_module.cc
firstAnswer01.fcl first.fcl
```

The file `CMakeLists.txt` in here contains yet more instructions for the build system and will be discussed later. The file `First_module.cc` is the first module that you will

look at and `first.fcl` is the FHiCL file that runs it. This exercise will suggest that you try to write some code on your own; the answer is provided in `FirstAnswer01_module.cc` and the file `firstAnswer01.fcl` runs it. These files will be discussed at length throughout the exercises.

10.4.3 "Build Window" Setup

Again, advanced users wanting to manage their own working directory may skip to Section 10.4.3.2.

10.4.3.1 Standard Procedure

Now go to your build window and do the following:

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Chapter 5.
3. `cd $ART_WORKBOOK_WORKING_BASE/<username>/\`
   `  workbook`
4. `mkdir build-prof`

   The `build-prof` directory will be your *build* directory.
5. `cd build-prof`
6. `source ../art-workbook/ups/setup_for_development \`
   `  -p $ART_WORKBOOK_QUAL`

   The space before the backslash is required here; there must be a space before the -p. The output from this command will tell you to take some additional steps; *do not do those steps.*

7. `buildtool`

   This step may take a few minutes.

Skip Section 10.4.3.2 and move on to Section 10.4.4.

10.4.3.2 Using Self-managed Working Directory

The steps in this procedure that are the same as for the "standard" procedure are explained in Section 10.4.4.

Now go to your build window and do the following:

1. Log in to the computer you chose in Section 8.3.

2. Follow the site-specific setup procedure; see Chapter 5.

3. Make a directory to hold the code that you will build and cd to it; this will be your *build* directory in your *build* window.

4. Make another directory, *outside of the heirarchy rooted at your build directory*, to hold output files created by the workbook exercises. (Don't cd to it.)

5. `ln -s <directory-for-output-files> output`

6. ```
   source <your-SOURCE-directory>/ups/setup_for_development \
     -p $ART_WORKBOOK_QUAL
   ```

   The space before the backslash is required here; there must be a space before the -p. The output from this command (Listing 10.1) will tell you to take some additional steps; *do not do those steps.*
   ```

7. buildtool

10.4.4 Examine Build Window Setup

Logging in and sourcing the site-specific setup script should be clear by now. Notice that next you are told to cd to the same `workbook` directory as in Step 4 of the instructions for the source window. From there, you make a directory in which you will run builds (your *build* directory), andcd to it. (The name `build-prof` can be any legal directory name but it is suggested here because this example performs a profile build; this is explained in Section 3.6.7). Figure 10.2 shows roughly what the build directory contains.

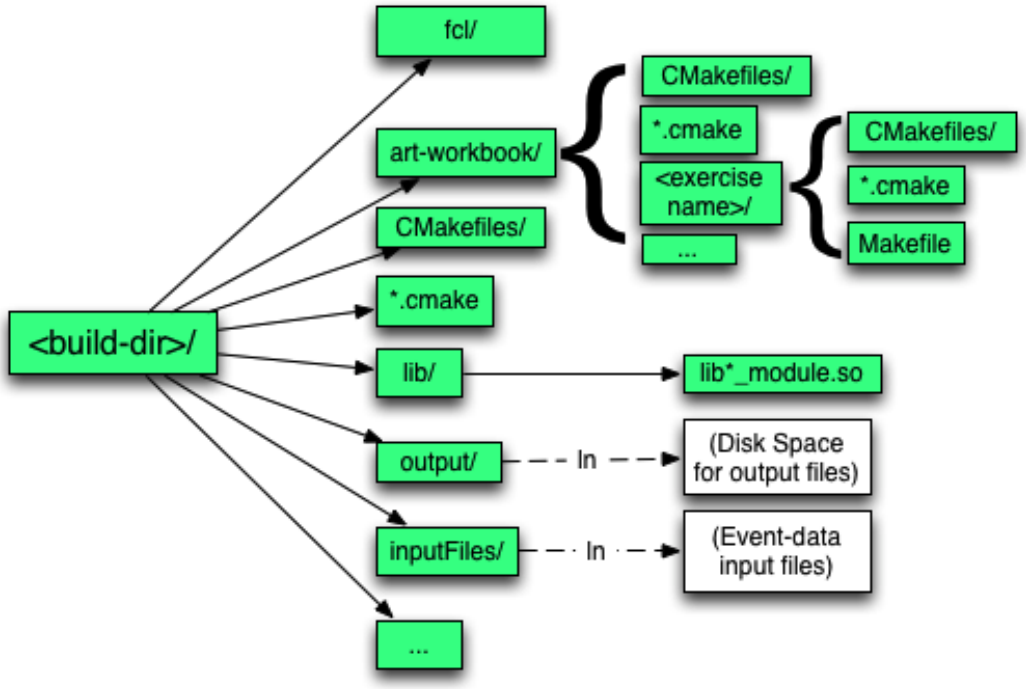


Figure 10.2: Representation of the reader's build directory structure (the `fcl/` directory is a symlink to `art-workbook/art-workbook/` in the source area)

Step 6 sources a script called `setup_for_development` found in the `ups` subdirec-

tory of the source directory. This script, run exactly as indicated, defines `build-prof` to be your build directory. This command selects a profile build (via the option -p); it also requests that the UPS qualifiers defined in the environment variable ART_WORKBOOK_QUAL be used when requesting the UPS products on which it depends; this environment variable was discussed in Section 9.9.3. The expected output is shown in Listing 10.1.

Check that there are no error messages in the indicated block. The listing concludes with a request for you to run a cmake command; *do not run* cmake (this line is an artifact of layering **cetbuildtools** on top of cmake).

Listing 10.1: Example of output created by `setup_for_development`

```
1
2  The working build directory is /ds50/app/user/kutschke/workbook/build-prof
3  The source code directory is /ds50/app/user/kutschke/workbook/art-workbook
4  ----------- check this block for errors ----------------------
5  -------------------------------------------------------------
6  /ds50/app/user/kutschke/workbook/build-prof/lib has been added to LD_LIBRARY_PATH
7  /ds50/app/user/kutschke/workbook/build-prof/bin has been added to PATH
8
9  CETPKG_SOURCE=/ds50/app/user/kutschke/workbook/art-workbook
10 CETPKG_BUILD=/ds50/app/user/kutschke/workbook/build-prof
11 CETPKG_NAME=art_workbook
12 CETPKG_VERSION=v0_00_15
13 CETPKG_QUAL=e2:prof
14 CETPKG_TYPE=Prof
15
16 Please use this cmake command:
17 cmake -DCMAKE_INSTALL_PREFIX=/install/path
18                             -DCMAKE_BUILD_TYPE=$CETPKG_TYPE $CETPKG_SOURCE
```

This script sets up all of the UPS products on which the Workbook depends; this is analogous to the actions taken by Step 6 in the first exercise (Section 9.6.1.1) when you were working in the *art* run-time environment. This script also creates several files and directories in your `build-prof` directory; these comprise the working space used by **cetbuildtools**.

After sourcing this script, the contents of `build-prof` will be

```
art_workbook-v0_00_18  bin lib
cetpkg_variable_report diag_report
```

At this time the two subdirectories `bin` and `lib` will be empty. The other files are used

by the build system to keep track of its configuration.

Step 7 (buildtool) tells **cetbuildtools** to build everything found in the source directory; this includes all of the Workbook exercises, not just the first one. The build process will take two or three mintues on an unloaded (not undergoing heavy usage) machine. Its output should end with the lines:

```
------------------------
INFO: Stage build successful.
------------------------
```

After the build has completed do an ls on the directory `lib`; you will see that it contains a large number of dynamic library (`.so`) files; for `v0_00_18` there will be about 30 `.so` files (subject to variation as versions change); these are the files that *art* will load as you work through the exercises.

Also do an ls on the directory `bin`; these are scripts that are used by **cetbuildtools** to maintain its environment; if the Workbook contained instructions to build any executable programs, they would have been written to this directory.

After runing buildtool, the build directory will contain:

```
admin                     CMakeFiles                fcl
art-workbook              cmake_install.cmake       inputFiles
art_workbook-v0_00_15     CPackConfig.cmake         lib
bin                       CPackSourceConfig.cmake   Makefile
cetpkg_variable_report    CTestTestfile.cmake       output
CMakeCache.txt            diag_report               ups
```

Most of these files are standard files that are explained in the **cetbuildtools** documentation, https://cdcvs.fnal.gov/redmine/projects/cetbuildtools/wiki . However, three of these items need special attention here because they are customized for the Workbook.

An ls -l on the files `fcl`, `inputFiles` and `output` will reveal that they are symbolic

links to

```
inputFiles -> ${TOYEXPERIMENT_DIR}/inputFiles
    output -> ${ART_WORKBOOK_OUTPUT_BASE}/
                                <username>/art_workbook_output
       fcl -> <your source directory>/art-workbook
```

These links are present so that the FHiCL files for the Workbook exercises do not need to be customized on a per-user or per-site basis.

- The link `inputFiles` points to the directory `inputFiles` present in the toyExperiment UPS product; this directory contains the input files that *art* will read when you run the first exercise. These are the same files used in the first exercise; if you need a reminder of the contents of these files, see Table 9.1. These input files will also be used in many of the subsequent exercises.
- The link `outputFiles` points to a directory that was created to hold your output files; the environment variable ART_WORKBOOK_OUTPUT_BASE was defined by your site-specific setup procedure.
- The symlink `fcl` points into your source directory hierarchy; it allows you to access the FHiCL files that are found in that hierarchy with the convenience of tab completions.

10.5 Setup for Subsequent Login Sessions

If you log out and later wish to log in again to work on this or any other subsequent exercise, you need to do the following:

In your source window:

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Table 5.1
3. cd to your source directory:

```
cd $ART_WORKBOOK_WORKING_BASE/<username>/\
  workbook/art-workbook
```

4. Set up the environment:

```
source ups/setup_deps -p
```

In your build window:

1. Log in to the computer you chose in Section 8.3.
2. Follow the site-specific setup procedure; see Chapter 5
3. cd to your build directory:

```
cd $ART_WORKBOOK_WORKING_BASE/<username>/\
  workbook/build-prof
```

4. Source the setup file:

```
source ../art-workbook/ups/setup_for_development \
  -p $ART_WORKBOOK_QUAL
```

If you chose to manage your own directory names, then the names of your source and build directories will be different than those shown.

Compare these steps with those given in Sections 10.4.1 and Section 10.4.3. You will see that some steps are missing from the source window and the build window instructions. The missing steps were only required the first time.

10.6 The *art* Development Environment

In the preceeding sections of this chapter you established what is known as the *art development environment*; this is a superset of the *art* run-time environment, which was described in Section 9.4. This section summarizes the new elements that are part of the development environment but not part of the run-time environment.

In Section 10.4.1, step 4b (git clone ...) contacted the central source code repository for the *art* Workbook code and made a clone of the repository in your source area under `art-workbook`; the clone contains the complete history of the repository, including all versions of art-workbook. Step 4d (git checkout ...) examined your clone of the repository, found the requested version of the code and put a working copy of that version into your source directory. The central repository is hosted on a Fermilab server and is accessed via the network. The upper left box in Figure 10.3 denotes the central repository and the box below it denotes the clone of the repository in your disk space; the box below that denotes the checked out working copy of the Workbook code. The flow of information during the clone and checkout processes is indicated by the green arrows (at left) in the figure.

In step 7 of Section 10.4.3, you ran buildtool in your build area, which read the source code files from your working copy of the Workbook code and turned them into dynamic libraries. The script `buildtool` is part of the build system, which is denoted as the box in the center left section of Figure 10.3. When you ran buildtool, it wrote dynamic library files to the `lib` subdirectory of your build directory; this directory is denoted in the figure as the box in the top center labeled `<build-directory>/lib`. The orange arrows in the figure denote the information flow at build-time. In order to perform this task, buildtool also needed to read header files and dynamic libraries found in the UPS products area, hence the orange arrow leading from the UPS Products box to the build system box.

In the figure, information flow at run-time is denoted by the blue lines. When you ran the *art* executable, it looked for dynamic libraries in the directories defined by LD_LIBRARY_PATH. In the *art* development environment, LD_LIBRARY_PATH contains

1. the `lib` subdirectory of your build directory
2. all of the directories previously described in Section 9.10

In all environments, the *art* executable looks for FHiCL files in

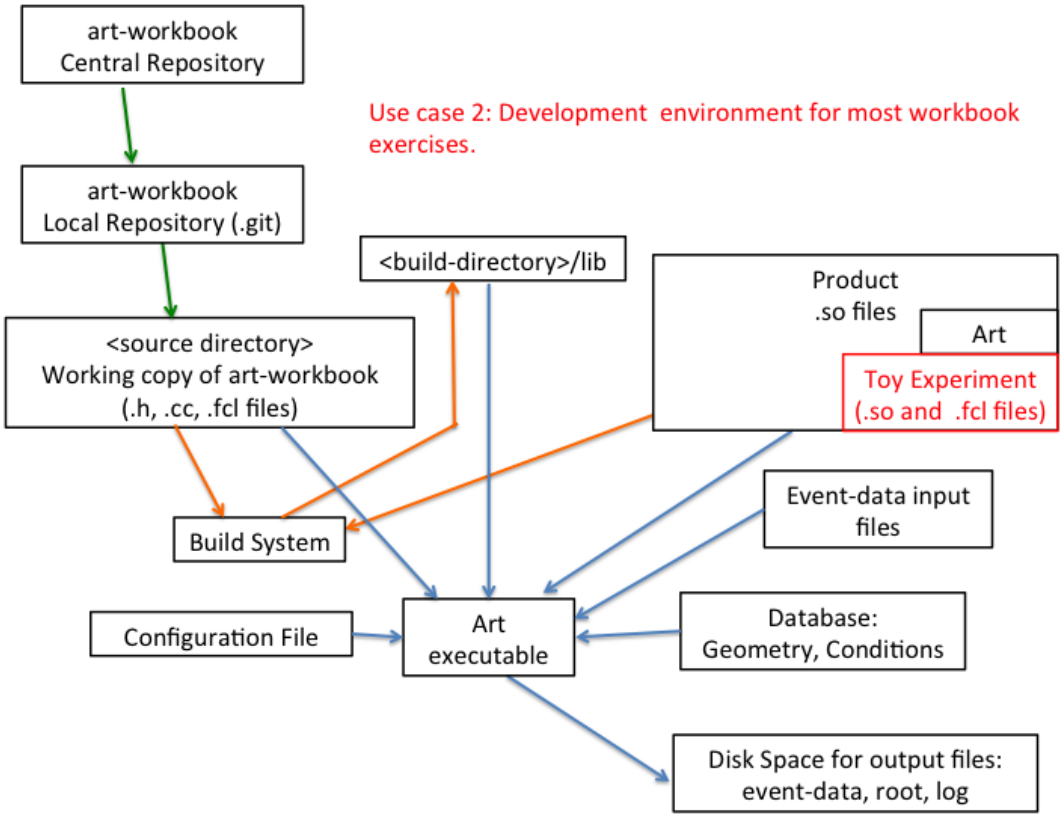


Figure 10.3: Elements of the *art* development environment as used in most of the Workbook exercises; the arrows denote information flow, as described in the text.

1. in the file specified in the -c command line argument
2. in the directories specified in FHICL_FILE_PATH

The first of these is denoted in the figure by the box labeled "Configuration File." In the *art* development environment, FHICL_FILE_PATH contains

1. some directories found in your checked out copy of the source
2. all of the directories previously described in Section 9.11

The remaining elements in Figure 10.3 are the same as described for Figure 9.1.

Figure 10.4, a combination of Figures 10.1 and 10.2), illustrates the distinct source and build areas, and the relationship between them. It does not show all the contents in each subdirectory.

10.7 Running the Exercise

10.7.1 Run *art* on `first.fcl`

In your build window, make sure that your current working directory is your build directory. From here, run the first part of this exercise by typing the following:

```
art -c fcl/FirstModule/first.fcl >& output/first.log
```

(As a reminder, we suggest you get in the habit of routing your output to the `output` directory.) The output of this step will look much like that in Listing 9.1, but with two signficant differences. The first difference is that the output from `first.fcl` contains an additional line

```
Hello from First::constructor.
```

The second difference is that the words printed out for each event are a little different; the printout from `first.fcl` looks like

```
Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1
```

while that from `hello.fcl` looked like

```
Hello World! This event has the id: run: 1 subRun: 0 event: 1
```

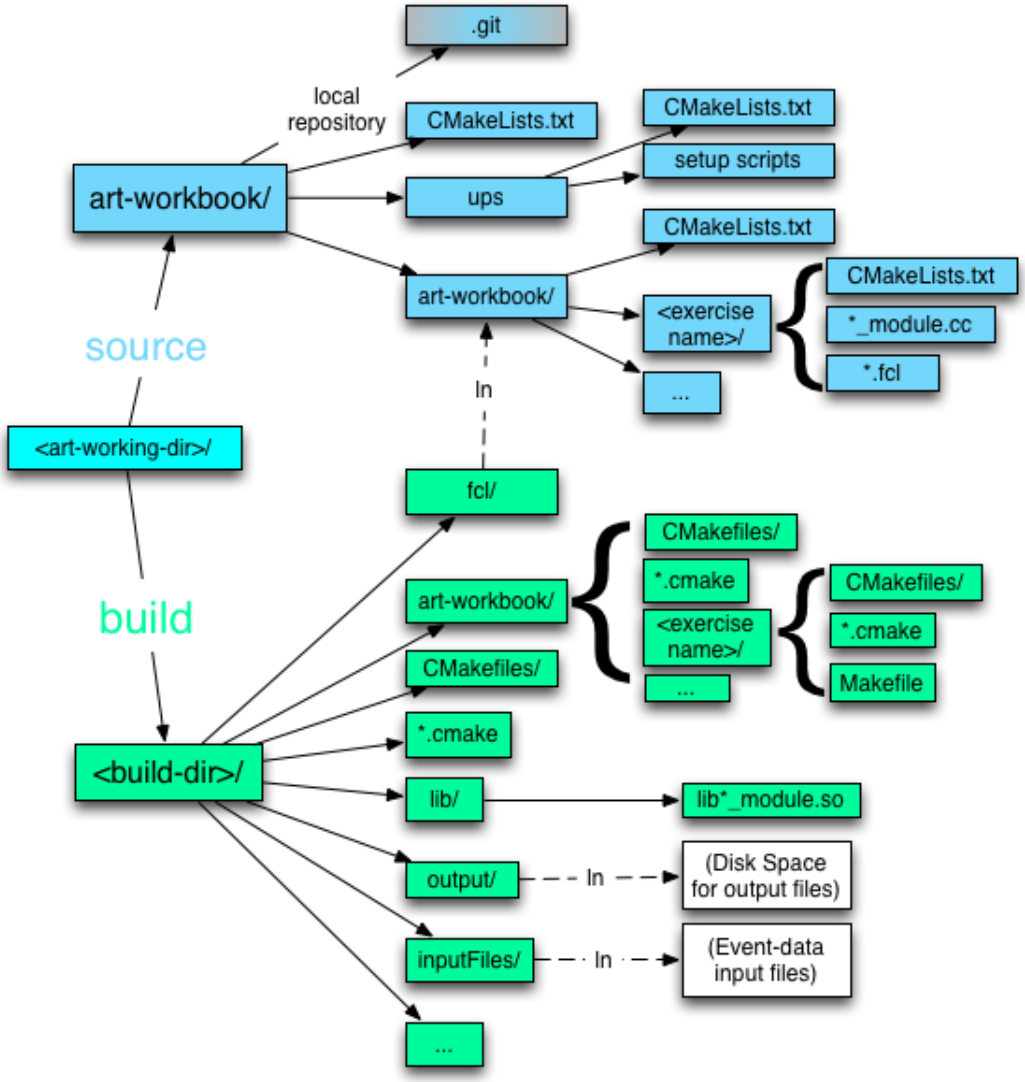


Figure 10.4: Representation of the reader's directory structure once the development environment is established.

The reason for changing this printout is so that you can identify, from the printout, which module was run.

10.7.2 The FHiCL File `first.fcl`

Compare the FHiCL file used in this exercise, `fcl/FirstModule/first.fcl`, with `hello.fcl` from the first exercise (i.e., run cat or diff on them). Other than comments, the only difference is that the `module_type` has changed from `HelloWorld` to `First`:

```
diff $TOYEXPERIMENT_DIR/HelloWorldScripts/hello.fcl fcl/FirstModule/first.fcl
```

```
...
<       module_type : HelloWorld
---
>       module_type : First
```

The file `first.fcl` tells *art* to run a module named `First`. As described in Section 9.10, *art* looks through the directories defined in LD_LIBRARY_PATH and looks for a file whose name matches the pattern `lib*First_module.so`. This module happens to be found at this location, relative to your build directory:

```
lib/libart-workbook_FirstModule_First_module.so
```

This dynamic library file was created when you ran buildtool.

10.7.3 The Source Code File `First_module.cc`

This section will describe the source code for the module `First` and will use it as a model to describe modules in general. The source code for this module is found in the following file, relative to your source directory (go to your source window!):

```
art-workbook/FirstModule/First_module.cc
```

When you ran buildtool, it compiled and linked this source file into the following dynamic library (relative to your your build directory):

```
lib/libart-workbook_FirstModule_First_module.so
```

This is the dynamic library that was loaded by *art* when you ran code for this exercise, in Section 10.7.2.

Look at the file `First_module.cc`, shown in Listing 10.2. In broad strokes, it does the following:

- declares a class named `First`
- provides the implementation for the class
- contains a call to the C-Preprocessor macro named DEFINE_ART_MODULE, discussed in Section 10.7.3.8

All module files that you will see in the Workbook share these "broad strokes." Some experiments that use *art* have chosen to split the source code for one module into three separate files; the *art* team does not recommend this practice, but it is in use and it will be discussed in Section 10.10.2.

10.7.3.1 The `#include` Statements

```
#include "art/Framework/Core/EDAnalyzer.h"
#include "art/Framework/Core/ModuleMacros.h"
#include "art/Framework/Principal/Event.h"
```

The first three lines of code in the file `First_module.cc` are *include directives* that pull in header files. All three of these files are included from the *art* UPS product (determining the location of included header files is discussed in Section 7.6).

```
#include <iostream>
```

The next line, `#include <iostream>`, includes the C++ header that enables this code to write output to the screen; for details, see any standard C++ documentation.

Those of you with some C++ experience may have noticed that there is no file named `First_module.h` in the directory `art-workbook/FirstModule`. The explanation for this will be given in Section 10.10.1.

Listing 10.2: The contents of `First_module.cc`

```
1
2  #include "art/Framework/Core/EDAnalyzer.h"
3  #include "art/Framework/Core/ModuleMacros.h"
4  #include "art/Framework/Principal/Event.h"
5
6  #include <iostream>
7
8  namespace tex {
9
10   class First : public art::EDAnalyzer {
11
12   public:
13
14     explicit First(fhicl::ParameterSet const& );
15
16     void analyze(art::Event const& event) override;
17
18   };
19
20 }
21
22 tex::First::First(fhicl::ParameterSet const& pset) : art::EDAnalyzer(pset) {
23   std::cout << "Hello␣from␣First::constructor." << std::endl;
24 }
25
26 void tex::First::analyze(art::Event const& event){
27   std::cout << "Hello␣from␣First::analyze.␣Event␣id:␣"
28             << event.id()
29             << std::endl;
30 }
31
32 DEFINE_ART_MODULE(tex::First)
```

If you are a C++ beginner you will likely find these header files difficult to understand; you do not need to understand them at this time but you do need to know where to find them for future reference.

10.7.3.2 The Declaration of the Class `First`, an Analyzer Module

Let's start with short explanations of each line and follow up with more information.

```
namespace tex {
```
Open a namespace named `tex`.

```
  class First : public art::EDAnalyzer{
```
- The first line of the declaration of the class `First`.
- Analyzer modules must inherit publicly from the base class EDAnalyzer.

```
  public:
```
Class members below here are public; any above would be private.

```
    explicit First(fhicl::ParameterSet const& );
```
- Declaration of a constructor.
- Its argument list is prescribed by *art*.
- *art* will call the constructor once at the start of each job.

```
    void analyze(art::Event const& event) override;
```
- Declaration of the `analyze` member function.
- Its argument list is prescribed by *art*.
- *art* will call this member function once per event.
- The *override* contextual identifier is an important safety feature. Use it!

```
  };
```
Close the declaration of the class `First`.

```
}
```
Close the namespace `tex`.

All of the code in the toyExperiment UPS product was written in the namespace `tex`; the name `tex` is an acronym-like shorthand for the toyExperiment (ToyEXperiment) UPS product. In order to keep things simple, all of the classes in the Workbook are also declared in the namespace `tex`. For more information about this choice, see Section 7.6.4. If you are not familiar with namespaces, consult the standard C++ documentation.

In the first line of the class declaration, the fragment "`: public art::EDAnalyzer`" tells the compiler that the class `First` is a *dervied class* that *inherits publicly* from the *base class* `art::EDAnalyzer`. At this time it is not necessary to understand inheritance, which is fortunate, because it takes a long, long time to explain. You just need to recognize and follow the pattern. You can read about C++ inheritance in the standard C++ documentation.

10.7.3.3 An Introduction to Analyzer Modules

Section 3.6.3 discussed the idea of *module types*: analyzer, producer, filter and so on. For a class to be a valid *art* analyzer module, it must follow a set of rules defined by *art*:

1. It must inherit publicly from `art::EDAnalyzer`.
2. It must provide a constructor with the argument list:
   `fhicl::ParameterSet const& pset`
   (Only the type of the argument is presecribed, not its name. You can use any name you want but the same name must be used in item 3.)
3. The initializer list of the constructor must call the constructor of the base class; and it must pass the parameter set to the constructor of the base class:
   `art::EDAnalyzer(pset)`
4. It must provide a member function named `analyze`, with the signature[¶]:
   `analyze( art::Event const&)`
5. If the name of a module class is `ClassName` then the source code for the module must be in a file named `ClassName_module.cc` and this file must contain the lines:
   `#include "art/Framework/Core/ModuleMacros.h"`
   and
   `DEFINE_ART_MODULE(namespace::ClassName)`

[¶] In C++ the *signature* of a member function is the name of the class of which the function is a member, the name of the function, the number, types and order of the arguments, and whether the member function is marked as `const or volatile. The signature does not include the return type; nor does it include the names of any of the arguments.`

6. It may optionally provide other member functions with signatures prescribed by *art*; if these member functions are present in a module class, then *art* will call them at the appropriate times. Some examples are provided in Chapter 12.

A module may also contain any other member data and any other member functions that are needed to do its job. You can see from Listing 10.2 that the class `First` follows all of the above rules and that it does not contain any of the optional member functions.

The requirement that the class name match the filename (minus the `_module.cc` portion) is enforced by *art*'s system for run-time loading of dynamic libraries. The requirement that the class provide the prescribed constructor is enforced by the macro `DEFINE_ART_MODULE`, which will be described in Section 10.7.3.8.

The declaration of the constructor begins with the keyword `explicit`. This is a safety feature this relevant only for constructors that have exactly one argument. A proper explanation would take too long so just follow a simple guideline: all constructors that have exactly one argument should be declared explicit. There will be rare circumstances in which you need to go against this guideline but you will not encounter any in the Workbook.

The *override* contextual identifier in the `analyzer` member function definition is a feature that is new in C++ 11 so older references will not discuss it. It is a new safety feature that we recommend you use; we cannot give a proper explanation until we have had a chance to discuss inheritance further. For now, just consider it a rule that, in all analyzer modules, you should provide this identifier as part of the declaration of `analyze`.

For those who are knowledgeable about C++, the base class `art::EDAnalyzer` declares the member function `analyze` to be pure virtual; so it must be provided by the derived class. The optional member functions of the base class are declared virtual but not pure virutal; do-nothing versions of these member functions are provided by the base class.

In a future version of this documentation suite, more information will be available in the Users Guide.

10.7.3.4 The Constructor for the Class `First`

The next code in the source file (Listing 10.2) is the *definition* of the constructor for the class `First`. This constructor simply prints some information (via `std::cout`) to let the user know that it has been called.

```
tex::First::First(fhicl::ParameterSet const& pset) : art::EDAnalyzer(pset) {
  std::cout << "Hello from First::constructor." << std::endl; }
```

The fragment `tex::First::First` says that this definition is for a constructor of the class `First` from the namespace `tex`.

The argument to the constructor is of type `fhicl::ParameterSet const&` as required by *art*. The class `ParameterSet`, found in the namespace `fhicl`, is a C++ representation of a FHiCL parameter set (aka FHiCL *table*). You will learn how to use this parameter set object in Chapter 13.

The argument to the constructor is passed by `const` reference, `const&`. This is a requirement specified by *art*; if you write a constructor that does not have exactly the correct argument type, then the compiler will issue a diagnostic and will stop compilation.

The first line of the constructor contains the fragment "`: art::EDAnalyzer(pset)`". This is the constructor's initializer list and it tells the compiler to call the constructor of the base class `art::EDAnalyzer`, passing it the parameter set as an argument. This is required by rule 3 in the list in Section 10.7.3.3.

The requirement that the constructor of an analyzer module pass the parameter set to the constructor of `art::EDAnalyzer` started in *art* version 1.08.09. If you are using an earlier version of *art*, constructors of analyzer modules must NOT call the constructor of `art::EDAnalyzer`.

10.7.3.5 Aside: Omitting Argument Names in Function Declarations

In the declaration of the class `First`, you may have noticed that the declaration of the member function `analyze` supplied a name for its argument (`event`) but the declaration

of the constructor did not supply a name for its argument.

In the declaration of a function, a name supplied for an argument is ignored by the compiler. So code will compile correctly with or without a name. Remember that a constructor is just a special kind of function so the rule applies to constructors too. It is very common for authors of code to provide an argument name as a form of documentation. You will code written both with and without named arguments in declarations.

The above discussion only applied to the *declarations* of functions, not to their definition (aka implementation).

10.7.3.6 The Member Function `analyze` and the Representation of an Event

The definition of the member function `analyze` comes next in the source file and is reproduced here

```
void tex::First::analyze(art::Event const& event){
  std::cout << "Hello from First::analyze. Event id: "
            << event.id()
            << std::endl;
}
```

If the type of the argument is not exactly correct, including the the `const&`, the compiler will issue a diagnostic and stop compilation. The compiler is able to do this because of one of the features of *inheritance*; the details of how this works is beyond the scope of this discussion.

Note that the `override` contextual identifier that was present in the declaration of this member function is not present in its definition; this is standard C++ usage.

Section 3.6.1 discussed the HEP idea of an event and the *art* idea of a three-part event identifier. The class `art::Event` is the representation within *art* of the HEP notion of an event. For the present discussion it is safe to consider the following over-simplified view of an event: it contains an event identifier plus a collection of data products (see Section 3.6.4). The name of the argument `event` has no meaning either to *art* or to the compiler — it is just an identifier — but your code will be easier to read if you choose a meaningful name.

At any given time in a running *art* program there is only ever one `art::Event` object; in the rest of this paragraph we will call this object *the event*. It is owned and managed by *art*, but *art* lets analyzer modules see the contents of the event; it does so by passing the event by `const` reference when it calls the `analyze` member function of analyzer modules. Because the event is passed by reference (indicated by the `&`), the member function `analyze` does not get a copy of the event; instead it is told where to find the event. This makes it efficient to pass an event object even if the event contains a lot of information. Because the argument is a `const` reference, if your code tries to change the contents of the event, the compiler will issue a diagnostic and stop compilation.

As described in Section 3.6.3, analyzer modules may only inspect data in `event`, not modify it. This section has shown how *art* institutes this policy as a hard rule that will be enforced rigorously by the compiler:

1. The compiler will issue an error if an analyzer module does not contain a member function named `analyze` with exactly the correct signature.
2. In the correct signature, the argument `event` is a `const` reference.
3. Because `event` is const, the compiler will issue an error if the module tries to call any member function of `art::Event` that will modify the event.

You can find the header file for `art::Event` by following the guidelines described in Section 7.6.2. *A future version of this documentation will contain a chapter in the Users Guide that provides a complete explanation of* `art::Event`. Here, and in the rest of the Workbook, the features of `art::Event` will be explained as needed.

The body of the function is almost trivial: it prints some information to let the user know that it has been called. In Section 10.7.1, when you ran *art* using `first.fcl`, the printout from the first event was

```
Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1
```

If you compare this to the source code you can see that the fragment `« event.id()` creates the following printout

```
run: 1 subRun: 0 event: 1
```

This fragment tells the compiler to do the following:

1. In the class `art::Event`, find the member function named `id()` and call this member function on the object `event`. This returns an object of type `art::EventID`, which is the class that represents an *art* event identifier, which was described in Section 3.6.1. You will learn more about `art::EventID` in Section 10.7.3.7.
2. Print the event identifier.

10.7.3.7 Representing an Event Identifier with `art::EventID`

Section 3.6.1 discussed the idea of an event identifier, which has three components, a run number, a subRun number and event number. In this section you will learn where to find the class that *art* uses to represent an event identifier. Rather than simply telling you the answer, this section will guide you through the process of discovering the answer for yourself.

Before you work through this section, you may wish to review Section 7.6 which discusses how to find header files.

In Section 10.7.3.6 you learned that the member function `art::Event::id()` returns an object that represents the event identifier. To see this for yourself, look at the header file for `art::Event`. Enter:

```
less $ART_INC/art/Framework/Principal/Event.h
```

or use one of the code browsers discussed in 7.6.2. In this file you will find the definition of the member function `id()`:[‖]

```
EventID
id() const {return aux_.id();}
```

The important thing to look at here is the return type, `EventID`; you do not need to (or want to) know anything about the data member `aux_`. If you look near the beginning of `Event.h` you will see that it has the line:

```
#include "art/Persistency/Provenance/EventID.h"
```

[‖] In C++, newlines are treated the same as any other white space; so this could have been written on a single line but the authors of `Event.h` have adopted a style in which return types are always written on their own line.

which is the header file that declares `EventID`. Look at this file, e.g.,

```
less $ART_INC/art/Persistency/Provenance/EventID.h
```

and find the declaration for `EventID`; you will see that the class `EventID` is within the namespace `art`, making its full name `art::EventID`. Near the top of the file you will also see the comments:

```
// An EventID labels an unique readout of the data
// acquisition system, which we call an ``event''.
```

Look again at `EventID.h`; you will see that it has accessor methods that permit you see the three components of the event identfier:

```
  RunNumber_t    run()    const;
  SubRunNumber_t subRun() const;
  EventNumber_t  event()  const;
```

Earlier in `EventID.h` the C++ *type*** `EventNumber_t` was defined as:

```
namespace art {
  typedef std::uint32_t EventNumber_t;
}
```

meaning that the event number is represented as a 32-bit unsigned integer. A *typedef* (γ) is a different name, or an alias, by which a type can be identified. If you are not familiar with the C++ concept of *typedef*, or if you are not familiar with the definite-length integral types defined by the `<cstdint>` header, consult any standard C++ documentation. If you dig deeper into the layers included in the `art::EventID` header, you will see that the run number and subRun number are also implemented as 32-bit unsigned integers.

The authors of *art* might have chosen an alternate definition of `EventNumber_t`

```
namespace art {
  typedef unsigned EventNumber_t;
}
```

The difference is the use of `unsigned` rather than `std::uint32_t`. This alternate

**In C++ the collective noun *type*, refers to both the built-in types, such as int and float, plus user defined types, which include classes, structs and typedefs.

version was not chosen because it runs the risk that some computers might consider this type to have a length of 32 bits while other computers might consider it to have a length of 16 or 64 bits. In the defintion that is used by *art*, an event number is guaranteed to be exactly 32 bits on all computers.

Why did the authors of *art* insert the extra level of indirection and not simply define the following member function inside `art::EventID`?

```
std::unit32_t event()  const;
```

The answer is that it makes it easy to change the definition of the type should that be necessary. If, for example, an experiment requires that event numbers be of length 64 bits, only one change is needed, followed by a recompilation.

It is good practice to use typedefs for every concept for which the underlying data type is not absolutely certain.

It is a very common, but not universal, practice within the HEP C++ community that typedefs that are used to give context-specific names to the C++ built-in types (`int`, `float`, `char`, etc.) end in `_t`.

One last observation about `EventID.h`. Near the top of this file you can find the following fragment, with a few lines omitted for clarity:

```
namespace art {
  std::ostream &
  operator<<(std::ostream & os, EventID const & iID);
}
```

This tells the compiler that the class `art::EventID` has a stream insertion operator (see Section 6.6.10). Because this operator exists, the compiler knows how to use `std::cout` to print an object of type `art::EventID`. You have already used this capability — near the end of Section 10.7.3.6 see the discussion of the line

```
 << event.id()
```

10.7.3.8 DEFINE_ART_MACRO: The Module Maker Macros

The final line in `First_module.cc`,

```
DEFINE_ART_MODULE(tex::First)
```

invokes a C preprocessor macro. This macro is defined in the header file that was pulled in by

```
#include "art/Framework/Core/ModuleMacros.h"
```

If you are not familiar with the C preprocessor, don't worry; you do not need to look under the hood. But if you would like to learn about it, consult any standard C++ reference.

The `DEFINE_ART_MODULE` macro instructs the compiler to put some additional code into the dynamic library made by buildtool. This additional code provides the glue that allows *art* to create instances of the class `First` without ever seeing the header or the source for the class; it only gets to see the `.so` or `.dylib` file and nothing else.

The `DEFINE_ART_MODULE` macro adds two pieces of code to the `.so` file. It adds a factory function that, when called, will create an instance of `First` and return a pointer to the base classes `art::EDAnalyzer`. In this way, *art* never sees the derived type of any analyzer module; it sees all analyzer modules via pointer to base. When *art* calls the factory function, it passes as an argument the parameter set specified in the FHiCL file for this module instance. The factory function passes this parameter set through to the constructor of `First`. The second piece of code put into the `.so` file is a static object that will be instantiated at load time; when this object is constructed, it will contact the *art* module registry and register the factory function under the name `First`. When the FHiCL file says to create a module of type `First`, *art* will simply call the registered factory function, passing it the parameter set defined in the FHiCL file. This is the last step in making the connection between the source code of a module and the *art* instantiation of a module.

10.7.3.9 Some Alternate Styles

C++ allows some flexibility in syntax, which can be seen as either powerful or confusing, depending on your level of expertise. Here we introduce you to a few alternate styles that you will need to recognize and may want to use.

Look at the `std::cout` line in the `analyze` method of Listing 10.2:

```
  std::cout << "Hello from First::analyze. Event id: "
            << event.id()
            << std::endl;
}
```

This could have been written:

```
  art::EventID id = event.id();
  std::cout << "Hello from First::analyze. Event id: "
            << id
            << std::endl;
```

This alternate version explicitly creates a temporary object of type `art::EventID`, whereas the original version created an *implicit* temporary object. When you are first learning C++ it is often useful to break down compound ideas by introducing *explicit temporaries*. However, the recommended best practice is to not introduce explicit temporaries unless there is a good reason to do so.

You will certainly encounter the first line of the above written in a different style, too, i.e.,

```
  art::EventID id(event.id());
```

Here `id` is initialized using *constructor syntax* rather than using *assignment syntax*. For almost all classes these two syntaxes will produce exactly the same result.

You may also see the argument list of the analyze function written a little differently,

```
void  analyze( const art::Event& );
```

instead of

```
void  analyze( art::Event const& );
```

The position of the `const` has changed. These mean exactly the same thing and the compiler will permit you to use them interchangeably. In most cases, small differences in the placement of the `const` identifier have very different meanings but, in a few cases, both variants mean the same thing. When C++ allows two different syntaxes that mean the same thing, this documentation suite will point it out.

Finally, Listing 10.3 shows the same information as Listing 10.2 but using a style in which the namespace remains open after the class declaration. In this style, the leading `tex::` is no longer needed in the definitions of the constructor and of `analyze`. Both layouts of the code have the same meaning to the compiler. Many experiments use this style in their source code.

Listing 10.3: An alternate layout for `First_module.cc`

```
1
2  #include "art/Framework/Core/EDAnalyzer.h"
3  #include "art/Framework/Core/ModuleMacros.h"
4  #include "art/Framework/Principal/Event.h"
5
6  #include <iostream>
7
8  namespace tex {
9
10   class First : public art::EDAnalyzer {
11
12   public:
13
14     explicit First(fhicl::ParameterSet const& );
15
16     void analyze(art::Event const& event) override;
17
18   };
19
20   First::First(fhicl::ParameterSet const& pset ) : art::EDAnalyzer(pset){
21     std::cout << "Hello␣from␣First::constructor." << std::endl;
22   }
23
24   void First::analyze(art::Event const& event){
25     std::cout << "Hello␣from␣First::analyze.␣Event␣id:␣"
26               << event.id()
27               << std::endl;
28   }
29
30 }
31
32 DEFINE_ART_MODULE(tex::First)
```

10.8 What does the Build System Do?

10.8.1 The Basic Operation

In Section 10.4.3 you issued the command buildtool, which *built* `First_module.so`. The purpose of this section is to provide some more details about building modules.

When you ran buildtool it performed the following steps:

1. It *compiled* `First_module.cc` to create an object file (ending in `.o`).
2. It *linked* the object file against the libraries on which it depends and inserted the result into a dynamic library (ending in `.so`).

The object file contains the machine code for the class `tex::First` and the machine code for the additional items created by the `DEFINE_ART_MODULE` C preprocessor macro. The dynamic library contains the information from the object file plus some additional information that is beyond the scope of this discussion. This process is called *building* the module.

The verb *building* can mean different things, depending on context. Sometimes is just means compiling; sometimes is just means linking; more often, as in this case, it means both.

To be complete, when you ran buildtool it built all of code in the Workbook, both modules and non-modules, but this section will only discuss how it built `First_module.so` starting from `First_module.cc`.

How did buildtool know what to do? The answer is that it looked in your source directory, where it found a file named `CMakeLists.txt`; this file contains instructions for **cetbuildtools**. Yes, when you ran buildtool in your build directory, it did look in your source directory; it knew to do this because, when you sourced `setup_for_development`, it saved the name of the source directory. The instructions in `CMakeLists.txt` tell **cetbuildtools** to look for more instructions in the subdirectory `ups` and in the file `art-workbook/CMakeLists.txt`, which, in turn, tells it to look for more instructions in the `CMakeLists.txt` files in each subdirectory of `art-workbook`.

When **cetbuildtools** has digested these instructions it knows the rules to build everything that it needs to build.

The object file created by the compilation step is a temporary file and, once it has been inserted into the dynamic library, it is not used any more. Therefore the name of the object file is not important.

On the other hand, the name of the dynamic library file is very important. *art* requires that for every module source file (ending in `_module.cc`) the build system must create exactly one dynamic library file (ending in `_module.so`). It also requires that the name of each `_module.so` file conform to a pattern. Consider the example of the file `First_module.cc`; *art* requires that the dynamic library for this file match the pattern

```
lib*First_module.so
```

where the * wildcard matches 0 or more characters.

When naming dynamic libraries, buildtool uses the following algorithm, which satisfies the *art* requirements and adds some addtional features; the algorithm is illustrated using the example of `First_module.cc`:

1. find the relative path to the source file, starting from the source directory
   `art-workbook/FirstModule/First_module.cc`
2. replace all slashes with underscores
   `art-workbook_FirstModule_First_module.cc`
3. change the trailing `.cc` to `.so`
   `art-workbook_FirstModule_First_module.so`
4. add the prefix lib
   `libart-workbook_FirstModule_First_module.so`
5. put the file into the directory `lib`, relative to the build directory
   `lib/libart-workbook_FirstModule_First_module.so`

You can check that this file is there by issuing the following command from your build directory:

```
ls -l lib/libart-workbook_FirstModule_First_module.so
```

This algorithm guarantees that every module within `art-workbook` will have a unique name for its dynamic library.

The experiments using *art* have a variety of build systems. Some of these follow the minimal *art*-conforming pattern, in which the wildcard is replaced with zero characters. If the Workbook had used such a build system, the name of the dynamic library file would have been

```
lib/libFirst_module.so
```

Both names are legal.

10.8.2 Incremental Builds and Complete Rebuilds

When you edit a file in your source area you will need to rebuild that file in order for those changes to take effect. If any other files in your source area depend on the file that you edited, they too will need to be rebuilt. To do this, reissue the command:

```
buildtool
```

Remember that this command must be executed from your build directory and that, before executing it, you must have setup the environment in your build window. When you run this command, **cetbuildtools** will automatically determine which files need to be rebuilt and will rebuild them; it will not waste time rebuilding files that do not need to be rebuilt. This is called an *incremental build* and it will usually complete much faster than the initial build.

If you want to clean up everything in your build area and rebuild everything from scratch, use the following command:

```
buildtool -c
```

This command will give you five seconds to abort it before it starts removing files; to abort, type ctrl-C in your build window. It will take about the same time to execute as did your initial build of the Workbook. The name of the option -c is a mnemonic for "clean".

When you do a clean build it will remove all files in your build directory that are not managed by **cetbuildtools**. For example, if you redirected the output of *art* as follows,

```
art -c fcl/FirstModule/first.fcl >& first.log
```

then, when you do a clean build, the file `first.log` will be deleted. This is why the instructions earlier in this chapter told you to redirect ouptut to a log file by

```
art -c fcl/FirstModule/first.fcl >& output/first.log
```

When you ran buildtool, it created a directory to hold your output files and you created a symbolic link, named output, from your build directory to this new directory. Both the other directory and the symbolic link survive clean builds and your output files will be preserved. The Workbook exercises write all of their root and event-data output files to this directory.

If you edit certain files in the `ups` subdirectory of your source directory, rebuilding requires an extra step. If you edit one of these files, the next time that you run buildtool, it will issue an error message saying that you need to re-source `setup_for_development`. If you get this message, make sure that you are in your build directory, and

```
source ../art-workbook/ups/setup_for_development \
  -p $ART_WORKBOOK_QUAL
```

```
buildtool
```

10.8.3 Finding Header Files at Compile Time

When `setup_for_development` establishes the working environment for the build directory, it does a UPS setup on the UPS products that it requires; this triggers a chain of additional UPS setups. As each UPS product is set up, that product defines many enviroment variables, two of which are <PRODUCT-NAME>_INC and <PRODUCT-NAME>_LIB. The first of these points to a directory that is the root of the header file hierarchy for that version of that UPS product. The second of these points to a single directory that holds all of the dynamic library files for that UPS product.

You can spot-check this by doing, for example,

```
ls $TOYEXPERIMENT_INC/*
```

```
ls $TOYEXPERIMENT_LIB
```

```
ls $ART_INC/*
```

```
ls $ART_LIB
```

You will see that the _INC directories have a subdirectory tree underneath them while the _LIB directories do not.

There are a few small perturbations on this pattern. The most visible is that the ROOT product puts most of its header files into a single directory, $ROOT_INC. The Geant4 product does the same thing.

When the compiler compiles a .cc file, it needs to know where to find the files specified by the #include directives. The compiler looks for included files by first looking for arguments on the command line, of the form

```
-I<path-to-a-directory>
```

There may be many such arguments on one command line. The compiler assembles the set of all -I arguments and uses it as an include path; that is, it looks for the header files by trying the first directory in the path and if it does not find it there, it tries the second directory in the path, and so on. The choice of -I for the name of the argument is a mnemonic for Include.

When buildtool compiles a .cc file it adds many -I options to the command line; it adds one for each UPS product that was set up when you sourced setup_for_development. When building First_module.cc, buildtool added -I$ART_INC, -I$TOYEXPERIMENT_INC and many more.

A corollary of this discussion is that when you wish to include a header file from a UPS product, the #include directive must contain the relative path to the desired file, starting from the _INC environment variable for that UPS product.

This system illustrates how the Workbook can work the same way on many different computers at many different sites. As the author of some code, you only need to know paths of include files relative to the relevant _INC environment variable. This environment variable may have different values from one computer to another but the setup and build systems will ensure that the site-specific information is communicated to the compiler using environment variables and the -I option.

This system has the potential weakness that if two products each have a header file with exactly the same relative path name, the compiler will get confused. Should this happen, the compiler will always choose the file from the earlier of the two -I arguments on the command line, even when the author of the code intended the second choice to be used. To

mitgate this problem, the *art* and UPS teams have adopted the convention that, whenever possible, the first element of the relative path in an `#include` directive will be the UPS package name. It is the implementation of this convention that led to the repeated directory name `art-workbook/art-workbook` that you saw in your source directory. There are a handful of UPS products for which this pattern is not followed and they will be pointed out as they are encountered.

The convention of having the UPS product name in the relative path of `#include` directives also tells readers of the code where to look for the included file.

10.8.4 Finding Dynamic Library Files at Link Time

The module `First_module.cc` needs to call methods of the class `art::Event`. Therefore the compiler left a notation in the object file saying "to use this object file you need to tell it where to find `art::Event`." The technical way to say this is that the object file contains a list of *undefined symbols* or *undefined external references*. When the linker makes the dynamic library

```
libart-workbook_FirstModule_First_module.so
```

it must resolve all of the undefined symbols from all of the object files that go into the library. To resolve a symbol, the linker must learn what dynamic library defines that symbol. When it discovers the answer, it will write the name of that dynamic library into something called the *dependency list* that is kept inside the dynamic library. **cetbuildtools** tells the linker that the dependency list should contain only the filename of each dynamic library, not the full path to it. If, after the linker has finished, there remain unresolved symbols, then the linker will issue an error message and the build will fail.

If library A depends on library B and library B depends on library C, but library A does not directly depend on library C, then the dependency list of library A should contain only library B. In other words, the dependency should contain only *direct dependencies* (also called *first order* dependencies).

To learn where to look for symbol definitions, the linker looks at its command line to find something called the *link list*. The link list can be specified in several different ways and the way that **cetbuildtools** uses is simply to write the link list as the absolute path to every `.so` file that the linker needs to know about. The link list can be different for every

dynamic library that the build system builds. However it is very frequently true that if a directory contains several modules, then all of the modules will require the same link list. The bottom line is that the author of a module needs to know the link list that is needed to build the dynamic library for that module.

For these Workbook exercises, the author of each exercise has determined the link list for each dynamic library that will be built for that exercise. In the **cetbuildtools** system, the link list for `First_module.cc` is located in the `CMakeLists.txt` file from same directory as `First_module.cc`; the contents of this file are shown in Listing 10.4. This `CMakeLists.txt` file says that all modules found in this directory should be built with the same link list and it gives the link list; the link list is the seven lines that begin with a dollar sign; these lines each contain one cmake variable. Recall that **cetbuildtools** is a build system that lives on top of cmake, which is another build system. A cmake variable is much like an environment variable except that is only defined within the environment of the running build system; you cannot look at it with printenv.

The five cmake variables beginning with ART_ were defined when buildtool set up the UPS *art* product. Each of these variables defines an absolute path to a dynamic library in $ART_LIB. For example ${ART_FRAMEWORK_CORE} resolves to

```
$ART_LIB/libart_Framework_Core.so
```

Almost all *art* modules will depend on these five libraries. Similarly the other two variables resolve to dynamic libraries in the **fhiclcpp** and **cetlib** UPS products.

When **cetbuildtools** constructs the command line to run the linker, it copies the link list from the `CMakeLists.txt` file to the command linker line.

The experiments that use *art* use a variety of build systems. Some of these build systems do not require that all external symbols be resolved at link time; they allow some external symbols to be resolved at run-time. This is legal but it can lead to certain difficulties. *A future version of this documentation suite will contain a chapter in the Users Guide that discusses linkage loops and how use of closed links can prevent them. This section will then just reference it.*

Consult the cmake and **cetbuildtools** documentation to understand the remaining details of this file.

Listing 10.4: The `file art-workbook/FirstModule/CMakeLists.txt`

```
1 art_make(MODULE_LIBRARIES
2   ${ART_FRAMEWORK_CORE}
3   ${ART_FRAMEWORK_PRINCIPAL}
4   ${ART_PERSISTENCY_COMMON}
5   ${ART_FRAMEWORK_SERVICES_REGISTRY}
6   ${ART_FRAMEWORK_SERVICES_OPTIONAL}
7   ${FHICLCPP}
8   ${CETLIB}
9   )
```

10.8.5 Build System Details

This section provides the next layer of details about the build system; *in a future version of this documentation set, the Users Guide will have a chapter with all of the details.* This entire section contains expert material.

If you want to see what buildtool is actually doing, you can enable verbose mode by issuing the command:

```
buildtool VERBOSE=TRUE
```

For example, if you really want to know the name of the object file, you can find it in the verbose output. For this exercise, the object file is

```
./art-workbook/FirstModule/CMakeFiles/
art-workbook_FirstModule_First_module.dir/First_module.cc.o
```

where the above is really just one line.

Also, you can read the verbose listing to discover the flags given to the compiler and linker. The more instructive compiler and linker flags valid at time of writing are given in Table 10.1.The C++ 11 features are selected by the presence of the `-std=c++11` flag and a high level of error checking is specified. The linker flag,

```
-Wl,--no-undefined
```

tells the linker that it must resolve all external references at link time. This is sometime referred to as a *closed link*.

Table 10.1: Compiler and linker flags for a profile build

Step	Flags
Compiler	-Dart_workbook_FirstModule_First_module_EXPORTS -DNDEBUG
Linker	-Wl,--no-undefined -shared
Both	-O3 -g -fno-omit-frame-pointer -Werror -pedantic -Wall -Werror=return-type -Wextra -Wno-long-long -Winit-self -Woverloaded-virtual -std=c++11 -D_GLIBCXX_USE_NANOSLEEP -fPIC

10.9 Suggested Activities

This section contains some suggested exercises in which you will make your own modules and learn more about how to use the class `art::EventID`.

10.9.1 Create Your Second Module

In this exercise you will create a new module by copying `First_module.cc` and making the necessary changes; you will build it using buildtool; you will copy `first.fcl` and make the necessary changes; and you will run the new module using the new FHiCL file.

Go to your source window and `cd` to your source directory. If you have logged out, out remember to re-establish your working environment; see Section 10.5 Type the following commands:

```
cd art-workbook/FirstModule
```

```
cp First_module.cc Second_module.cc
```

```
cp first.fcl second.fcl
```

Edit the files `Second_module.cc` and `second.fcl`. In both files, change every occurence of the string “First” to “Second”; there are eight places in the source file and two in the FHiCL file, one of which is in a comment.

The new module needs the same link list as did `First_module.cc` so there is no need to edit `CMakeLists.txt`; the instructions in `CMakeLists.txt` tell buildtool to build

all modules that it finds in this directory and to use the same link list for all modules.

Go to your build window and cd to your build directory. Again, remember to re-establish your working environment as necessary. Rebuild the Workbook code:

```
buildtool
```

This should complete with the message:

```
------------------------------------
INFO: Stage build successful.
------------------------------------
```

If you get an error message, consult a local expert or the *art* team as described in Section 3.4.

When you run buildtool it will perform an incremental build (see Section 10.8.2) during which it will detect `Second_module.cc` and build it.

You can verify that buildtool created the expected dynamic library:

```
ls lib/*Second*.so
```

```
lib/libart-workbook_FirstModule_Second_module.so
```

Stay in your build directory and run the new module:

```
art -c fcl/FirstModule/second.fcl >& output/second.log
```

Compare `output/second.log` with `output/first.log`. You should see that "First" has been replaced by "Second" everywhere and the date/time lines are different.

10.9.2 Use *artmod* to Create Your Third Module

This exercise is much like the previous one; the difference is that you will use a tool named artmod to create the source file for the module.

Go to your source window and cd to your source directory. If you have logged out, remember to re-establish your working environment; see Section 10.5

The command artmod creates a file containing the skeleton of a module. It is supplied by the UPS product **cetpkgsupport**, which was set up when you performed the last step

of establishing the environment in the source window, sourcing `setup_deps`. You can verify that the command is in your path by using the bash built-in command type (output shown on two lines):

```
type artmod
```

```
artmod is hashed (/ds50/app/products/cetpkgsupport/
    v1_02_00/bin/artmod)
```

The leading elements of the directory name will reflect your UPS products area, and may be different from what is shown here. The version number, `v1_02_00`, may also change with time.

From your source directory, type the following commands:

```
cd art-workbook/FirstModule
```

```
artmod analyzer tex::Third
```

```
cp first.fcl third.fcl
```

The second command tells artmod to create a source file named `Third_module.cc` that contains the skeleton for an 'analyzer' module, to be named `Third` in the namespace `tex`.

If you compare `Third_module.cc` to `First_module.cc` you will see a few differences:

1. `Third_module.cc` is longer and has more comments
2. the layout of the class is a little different but the two layouts are equivalent
3. there are some extra `#include` directives
4. the include for `<iostream>` is missing
5. in the `analyze` member function, the name of the argument is different (`event` vs `e`)
6. artmod supplies the skeleton of a destructor (`~Third`)

The `#include` directives provided by artmod are a best guess, made by the author of artmod, about which ones will be needed in a "typical" module. Other than slowing down

the compiler by an amount you won't notice, the extra `#include` directives do no harm; keep them or leave them as you see fit.

Edit `Third_module.cc`

1. add the `#include` directive for `<iostream>`
2. copy the bodies of the constructor and the `analyze` member function from `First_module.cc`; change the string "First" to "Third"
3. in the definition of the member function `analyze`, change the name of the argument to `event`.

When you built `First_module.cc`, the compiler wrote a destructor for you that is identical to the destructor written by artmod; so you can leave the destructor as artmod wrote it, i.e., with an empty body. Or you can delete it; if you decide to do so, you must delete both the declaration and the implementation.

Edit `third.fcl` Change every occurence of the string "First" to "Third"; there are two places, one of which is in a comment.

Go to your build window and cd to your build directory. If you have logged, out remember to re-establish your working environment; see Section 10.5. Rebuild the Workbook code:

```
buildtool
```

Refer to the previous section to learn how to identify a successful build and how to verify that the expected library was created.

Stay in your build directory and run the third module:

```
art -c fcl/FirstModule/third.fcl >& output/third.log
```

Compare `output/third.log` with `output/first.log`. You should see that the printout from `First_module.cc` has been replaced by that from `Third_module.cc`.

artmod has many options that you can explore by typing:

```
artmod --help
```

10.9.3 Running Many Modules at Once

In this exercise you will run four modules at once, the three made in this exercise plus the `HelloWorld` module from Chapter 9.

Go to your source window and cd to your source directory. Type the following commands:

```
cd art-workbook/FirstModule
```

```
cp first.fcl all.fcl
```

Edit the file `all.fcl` and replace the `physics` parameter set with the contents of Listing 10.5. This parameter set:

1. defines four module labels and
2. puts all four module labels into the `end_paths` sequence.

When you run *art* on this FHiCL file, *art* will first look at the definition of `end_paths` and learn that you want it to run four module labels. Then it will look in the `analyzers` parameter set to find the definition of each module label; in each definition *art* will find the class name of the module that it should run. Given the class name and the environment variable LD_LIBRARY_PATH, *art* can find the right dynamic library to load. If you need a refresher on module labels and `end_paths`, refer to Sections 9.8.7 and 9.8.8.

Go to your build window and cd to your build directory. If you have logged out, remember to re-establish your working environment; see Section 10.5. You do not need to build any code for this exercise.

Run the exercise:

```
art -c fcl/FirstModule/all.fcl >& output/all.log
```

Compare `output/all.log` with the log files from the previous exercises. The new log file should contain printout from each of the four modules. Once, near the start of the file, you should see the printout from the three constructors; remember that the `HelloWorld` module does not make any printout in its constructor. For each event you should see the printout from the four `analyze` member functions.

Listing 10.5: The `physics` parameter set for `all.fcl`

```
physics :{
  analyzers: {
    hello : {
      module_type : HelloWorld
    }
    first : {
      module_type : First
    }
    second : {
      module_type : Second
    }
    third : {
      module_type : Third
    }
 }

  e1        : [ hello, first, second, third ]
  end_paths : [ e1 ]

}
```

Remember that *art* is free to run analyzer modules in any order; this was discussed in Section 9.8.8.

10.9.4 Access Parts of the EventID

In this exercise, you will access the individual parts of the event identifier.

Before proceeding with this section, review the material in Section 10.7.3.7 which discusses the class `art::EventID`. The header file for this class is:

```
$ART_INC/art/Persistency/Provenance/EventID.h
```

In this exercise, you are asked to rewrite the file `Second_module.cc` so that the printout made by the analyze method looks like the following (lines split here due to space restrictions):

```
Hello from FirstAnswer01::analyze.  run number: 1
     sub run number: 0 event number: 1
Hello from FirstAnswer01::analyze.  run number: 1
```

```
sub run number: 0 event number: 2
```

and so on for each event.

To do this, you will need to reformat the text in the `std::cout` statement and you will need to separately extract the run, subRun and event numbers from the `art::EventID` object.

You will do the editing in your source window, in the subdirectory `art-workbook/FirstModule`.

When you think that you have successfully rewritten the module, you can test it by going to your build window and cd'ing to your build directory. Then:

```
buildtool
```

```
art -c fcl/FirstModule/second.fcl >& output/eventid.log
```

If you have not figured out how to do this exercise after about 15 minutes, you can find one possible answer in the file `FirstAnswer01_module.cc`, in the same directory as `First_module.cc`.

To run the answer module and verify that it makes the requested output, run:

```
art -c fcl/FirstModule/firstAnswer01.fcl >& output/firstAnswer01.log
```

(The command can be typed on a single line.) You did not need to build this module because it was already built the first time that you ran buildtool; that run of buildtool built all of the modules in the Workbook.

There is a second correct answer to this exercise. If you look at the header file for `art::Event`, you will see that this class also has member functions

```
EventNumber_t    event()   const {return aux_.event();}
SubRunNumber_t   subRun()  const {return aux_.subRun();}
RunNumber_t      run()     const {return id().run();}
```

So you could have called these directly,

```
std::cout << "Hello from FirstAnswer01::analyze. "
          << " run number: "     << event.run()
          << " sub run number: " << event.subRun()
          << " event number: "   << event.event()
          << std::endl;
```

instead of

```
std::cout << "Hello from FirstAnswer01::analyze. "
          << " run number: "     << event.id().run()
          << " sub run number: " << event.id().subRun()
          << " event number: "   << event.id().event()
          << std::endl;
```

But the point of this exercise was to learn a little about how to dig down into nested header files to find the information you need.

10.10 Final Remarks

10.10.1 Why is there no `First_module.h` File?

When you performed the exercises in this chapter, you saw, for example, the file `First_module.cc` but there was no corresponding `First_module.h` file. This section will explain why.

In a typical C++ programming environment there is a header file (`.h`) for each source file (`.cc`). As an example, consider the files `Point.h` and `Point.cc` that you saw in Section 6.6.10.

The reason for having `Point.h` is that the implementation of the class, `Point.cc`, and the users of the class need to agree on what the class `Point` is. In the Section 6.6.10 example, the only user of the class is the main program, `ptest.cc`. The file `Point.h` serves as the unique, authoritative declaration of what the class is; both `Point.cc` and `ptest.cc` rely on on this declaration.

If you think carefully, you are already aware of a very common exception to the pattern of one `.h` file for each `.cc` file: there is never a header file for a main program. For example,

in the examples that exercised the class `Point`, `ptest.cc` had no header file. Why not? No other piece of user-written code needs to know about any classes or functions declared or defined inside `ptest.cc`.

The `First_module.h` file is omitted simply because every entity that needs to see the declaration of the class `First` is already inside the file `First_module.cc`. There is no reason to have a separate header file. Recall the "dangerous bend" paragraph at the end of Section 10.7.3.8 that described how *art* is able to use modules without needing to know about the declaration of the module class.

art is designed such that only *art* may construct instances of module classes and only *art* may call member functions of module classes. In particular, modules may not construct other modules and may not call member functions of other modules. The absence of a `First_module.h`, provides a physical barrier that enforces this design.

10.10.2 The Three-File Module Style

In this chapter, the source for the module `First` was written in a single file. You may also write it using three files, `First.h`, `First.cc` and `First_module.cc`.

Some experiments use this three-file style. The authors of *art* do not recommend it, however, because it exposes the declaration of `First` in a way that permits it to be misused (as was discussed in Section 10.10.1). The build system distributed with the Workbook has not been configured to build modules written in this style.

In this style, `First.h` contains the class declaration plus any necessary `#include` directives; it now also requires code guards; this is shown in Listing 10.6.

Listing 10.6: The contents of `First.h` in the three-file model

```
1
2  #ifndef art-workbook_FirstModule_First_h
3  #define art-workbook_FirstModule_First_h
4
5  #include "art/Framework/Core/EDAnalyzer.h"
6  #include "art/Framework/Principal/Event.h"
7
8  namespace tex {
9
10   class First : public art::EDAnalyzer {
11
```

```
12   public:
13
14     explicit First(fhicl::ParameterSet const& );
15
16     void analyze(art::Event const& event) override;
17
18   };
19
20 }
21 #endif
```

The file `First.cc` contains the definitions of the constructor and the `analyze` member function, plus the necessary `#include` directives; this is shown in Listing 10.7.

Listing 10.7: The contents of `First.cc` in the three-file model

```
1
2  #include "art-workbook/FirstModule/First.h"
3
4  #include <iostream>
5
6  tex::First::First(fhicl::ParameterSet const& pset ) : art::EDAnalyzer(pset) {
7    std::cout << "Hello␣from␣First::constructor." << std::endl;
8  }
9
10 void tex::First::analyze(art::Event const& event){
11   std::cout << "Hello␣from␣First::analyze.␣Event␣id:␣"
12             << event.id()
13             << std::endl;
14 }
```

And `First_module.cc` is now stripped down to the invocation of the DEFINE_ART_MODULE macro plus the necessary `#include` directives; this is shown in Listing 10.8.

Listing 10.8: The contents of `First_module.cc` in the three-file model

```
1
2  #include "art-workbook/FirstModule/First.h"
3  #include "art/Framework/Core/ModuleMacros.h"
4
5  DEFINE_ART_MODULE(tex::First)
```

10.11 Flow of Execution from Source to FHiCL File

The properties that a class must have in order to be an analyer module are summarized in Section 10.7.3.2 for reference. This section reviews how the source code found in an analyzer module, e.g., `First_module.cc`, is executed by *art*:

1. The script `setup_for_development` defines many environment variables that are used by buildtool, *art* and toyExperiment.
2. LD_LIBRARY_PATH, an important environment variable, contains the directory `lib` in your build area plus the `lib` directories from many UPS products, including *art*.
3. buildtool compiles `First_module.cc` to a temporary object file.
4. buildtool links the temporary object file to create a dynamic library in the `lib` subdirectory of your build area:
   `lib/libart-workbook_FirstModule_First_module.so`
5. When you run *art* using file `first.fcl`, this file tells *art* to find and load a module with the "module_type" `First`.
6. In response to this request, *art* will search the directories in LD_LIBRARY_PATH to find a dynamic library file whose name matches the pattern:
   `lib*First_module.so`
7. If *art* finds either zero or more than one match to this pattern, it will issue an error message and stop.
8. If *art* finds exactly one match to this pattern, it will load the dynamic library.
9. After *art* has loaded the dynamic library, it has access to a function that can, on demand, create instances of the class `First`.

The last bullet really means that the dynamic library contains a factory function that can construct instances of `First` and return a pointer to the base class, `art::EDAnalyzer`. The dynamic library also contains a static object that, at load-time, will contact the *art* module registry and register the factory function under the `module_type First`.

11 Keeping Up to Date with Workbook Code and Documentation

11.1 Introduction

As you well know by now, the Workbook exercises require you to download some code to edit, build, execute and evaluate. Both the documentation and the code it references are expected to undergo continual development throughout 2014. The latest is always available at the *art* Documentation website.

Announcements of new releases are made on the `art-users@fnal.gov` mailing list. Please subscribe!

Until the full set of exercises is written, you will have to update occasionally just to get the latest exercises. Come back to this chapter whenever you reach the end of the available exercises. Or come back and update whenever a new release is announced; it may include improvements to existing exercises.

11.2 Special Instructions for Summer 2014

Summer 2014: Until further notice, if you need to obtain updated Workbook code, you will need to reinstall the Workbook code from scratch. The procedures below will usually work but there are some circumstances in which they won't. Until the workbook team can document how you should deal with the exceptional cases, please reinstall from scratch. To do so, use the following procedure:

1. Save your existing work so that you can refer to it later.
   (a) Go to the directory that is two above your source and build directories and get a directory listing:
   ```
   cd $ART_WORKBOOK_WORKING_BASE/<username>
   ls
   ```
   (b) You should see a directory named `workbook` that contains your source and build directories `art-workbook` and `build-prof`. You may also see other files and directories.
   (c) Choose a new name for the `workbook` directory, perhaps `workbook_sav1`. The suffix `_sav1` is just a suggestion — the only requirement is that the new name not conflict with existing ones.
   (d) Rename the `workbook` directory
   ```
   mv workbook workbook_sav1
   ```

2. Follow the instructions to install the Workbook code from scratch in Section 10.4.

11.3 How to Update

This chapter will show you how to update. The steps include:

For the moment, please restart from scratch. See the previous section

1. Determine whether an updated release is available, and what release it is.

2. Switch to the updated documentation.

3. In your source window, use git to update your working version of the code in the (higher-level) `art-workbook` directory

4. In your build window, build the new version of the code.

11.3.1 Get Updated Documentation

First, check which documentation release you're currently using: it's noted on the title page of this document*. Then go to the *art* Documentation website and compare your documentation release number to the latest available.

Download a new copy of the documentation, as needed.

11.3.2 Get Updated Code and Build It

Also noted on the title page of the documentation is the release† of the art-workbook code that the documentation is intended for. Recall from Figure 10.1 that git commands are used to clone the code in the remote repository into your local copy, then copy the requested release from that local copy into your working area. The git system is described in more detail in Chapter 19.

Chances are that you're using the code release that goes with the documentation you have been using. You can check by looking in the file `art-workbook/ups/product_deps`. From your source directory run:

```
grep art_workbook ups/product_deps
```

```
parent art_workbook v0_00_13
```

This shows version v0_00_13 as an example. If your version is earlier than the one listed on the cover of the latest documentation, you will need to get new code and build it.

These instructions illustrate updating the working version of the art-workbook code from version v0_00_13 to version v0_00_15. There is nothing special about these two versions; the instructions serve as a model for a change between any pair of versions.

1. Start from (or cd to) your source directory (see Section 10.4.1):
   ```
   cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/art-workbook
   ```

*Versions of the *art* documentation prior to 0.51 do not have this information on the front page; for these versions, the required version of art-workbook can be found in the section "Setting up to Run Exercises" in Exercise 2.

†The terms "release" and "version" are used interchangeably here.

2. Use git status and make a note of the files that you have modified and/or added (see Section 19.1.3 for instructions).
   ```
   git status [-s]
   ```

3. Switch from your tagged version branch back to the develop branch ("branches" are discussed in Chapter 19, you don't need to understand them at this stage[‡]).

   ```
   git checkout develop
   ```

   ```
   Switched to branch 'develop'
   ```

4. Update your local copy of the respository (the `.git` directory)

   ```
   git pull
   ```

   The output from this command is shown in Listing 11.1.

5. Switch your working code to the new branch:
   ```
   git checkout -b v0_00_15 v0_00_15
   ```

   ```
   Switched to a new branch 'v0_00_15'
   ```

   Use the new version number twice in this command. In the messages produced in this step, watch for the names of files that you have modified. Check for conflicts that git did not merge correctly.

To rebuild your updated working code:

1. In your build window, cd to your build directory
   ```
   cd $ART_WORKBOOK_WORKING_BASE/<username>/workbook/build-prof
   ```

2. Tell **cetbuildtools** to look for, and act on, any changes in your checked out version of the code (command shown on two lines):
   ```
   source ../art-workbook/ups/setup_for_development \-p $ART_WORKBOOK_QUAL
   ```

[‡]If you are familiar with git concepts, you may want to know this: The authors of the art-workbook follow the convention that they make a new git branch for every release of art-workbook and the name of the branch matches the version number of art-workbook. In the current example, the local working environment knows about two branches, the develop branch and the branch for version v0_00_13. The develop branch is the name of the branch that always contains the most recent art-workbook code.

Listing 11.1: Example of the output produced by git pull

```
From http://cdcvs.fnal.gov/projects/art-workbook
   e79d9ef..81d2a76  develop    -> origin/develop
   6435ecc..c0c1af5  master     -> origin/master
From http://cdcvs.fnal.gov/projects/art-workbook
 * [new tag]         v0_00_14   -> v0_00_14
 * [new tag]         v0_00_15   -> v0_00_15
Updating e79d9ef..81d2a76
Fast-forward
 art-workbook/ModuleInstances/magic.fcl       | 26 ++++++++++---------
 art-workbook/ParameterSets/PSet01_module.cc  | 36 +++++++++++++++-----------
 art-workbook/ParameterSets/PSet02_module.cc  | 53 +++++++++++++--------------------------
 art-workbook/ParameterSets/PSet03_module.cc  | 28 ++++++++++-----------
 art-workbook/ParameterSets/PSet04_module.cc  | 44 +++++++++++++++------------------
 art-workbook/ParameterSets/pset01.fcl        |  6 ++---
 art-workbook/ParameterSets/pset02.fcl        | 14 +++++++----
 art-workbook/ParameterSets/pset03.fcl        |  6 ++---
 art-workbook/ParameterSets/pset04.fcl        |  7 +++---
 ups/product_deps                             |  2 +-
 10 files changed, 109 insertions(+), 113 deletions(-)
```

3. Rebuild:
   `buildtool`
   If this step does not complete successfully, the first thing to try is a clean rebuild:
   `buildtool -c`

11.3.3 See which Files you have Modified or Added

At any time you can check to see which files you have modified and which you have added. The code is structured in such a way that when you checkout a new version, these files will remain in your working directory and will not be modified or deleted. The git checkout command will generate some informational messages about them, but you do not need to take any action.

To see the new/modified files, cd to your source directory and issue the `git status` command. Suppose that you have checked out version `v0_00_13`, modified `first.fcl` and added `second.fcl`. The git status command will produce the following output:

git status

```
# On branch v0_00_13
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
# modified:   first.fcl
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# second.fcl
no changes added to commit (use "git add" and/or "git commit -a")
```

Do not issue the git add or git commit commands that are suggested in the command output above.

In the rare case that you have neither modified nor added any files, the output of git status will look like:

git status

```
# On branch v0_00_13
```

12 Exercise 3: Some other Member Functions of Modules

12.1 Introduction

Recall the discussion in Section 3.6.2 about widget-making workers on an assembly line. All workers have a task to perform on each widget as it passes by and some workers may also need to perform start-up or shut-down tasks. If a module has something that it must do at the start of the job, then the author of the module can write a member function named `beginJob()` that performs these tasks. Similarly the author of a module can write a member function named `endJob` to do tasks that need to be performed at the end of the job. *art* will call both of these member functions at the appropriate time.

The author of a module may also provide member functions to perform actions at the start of a subRun, at start of a run, at the end of a subRun or at the end of a Run.

These member functions are *optional*; i.e., they are always allowed in a module but never required. They have prescribed names and argument lists.

In this exercise you will build and execute an analyzer module that illustrates three of these member functions: `beginJob`, `beginRun` and `beginSubRun`. These member functions are called, respectively, once at the start of the *art* job, once for each new run and once for each new subRun.

You may also perform a suggested exercise to add the three corresponding member functions `endJob`, `endRun` and `endSubRun`.

12.2 Prerequisites

The prerequisites for this chapter include all of the material in Part I (Introduction) and all of the material up to this point in Part II (Workbook).

In particular, make sure that you understand the *event loop* (see Section 3.6.2).

12.3 What You Will Learn

This chapter will show you *how* to provide the optional member functions in your *art* modules to execute special functionality at the beginning and end of jobs, runs and/or subRuns. These include

1. `beginJob()`
2. `beginRun( art::Run const&)`
3. `beginSubRun( art::SubRun const&)`
4. `endJob()`
5. `endRun( art::Run const&)`
6. `endSubRun( art::SubRun const&)`

As you gain experience, you will gain proficiency at knowing *when* to provide them.

You will also be introduced to the classes

1. `art::RunID`
2. `art::Run`
3. `art::SubRunID`
4. `art::SubRun`

that are analogous to the `art::EventID` and `art::Event` classes that you have already encountered.

12.4 Setting up to Run this Exercise

Follow the instructions in Section 10.5 if you are logging in after having closed an earlier session. If you are continuing on directly from the previous exercise, keep both your source and build windows open.

12.5 Files Used in this Exercise

In your source window, look at the contents of the directory for this exercise, called `OptionalMethods`:

```
ls art-workbook/OptionalMethods
```

```
CMakeLists.txt          OptionalAnswer01_module.cc
Optional_module.cc    optionalAnswer01.fcl    optional.fcl
```

The source code for the module you will run is `Optional_module.cc` and the FHiCL file to run it is `optional.fcl`. The file `CMakeLists.txt` is identical to that used by the previous exericse since the new features introduced by this module do not require any modifications to the link list. The other two files relate to the exercise you will be asked to do in Section 12.9.

In your build window, just make sure that you are in your build directory. All the code for this exercise is already built; this happened the first time that your ran buildtool.

12.6 The Source File `Optional_module.cc`

In your source window, look at the source file `Optional_module.cc` and compare it to `First_module.cc`. The differences are

1. it has two new include directives, for `Run.h` and `SubRun.h`
2. the name of the class has changed from `First` to `Optional`
3. the `Optional` class declaration declares three new member functions
   `void beginJob () override;`

```
void beginRun ( art::Run const& run ) override;
void beginSubRun( art::SubRun const& subRun ) override;
```

4. the text printed by the constructor and analyze member functions has changed
5. the file contains the definitions of the three new member functions, each of which simply makes some identifying printout

12.6.1 About the `begin*` Member Functions

The optional member functions `beginJob`, `beginRun` and `beginSubRun`, described in the Introduction to this chapter (Section 12.1), must have exactly the argument list prescribed by *art* as shown in list item 3 above.

art knows to call the `beginJob` member function of each module, if present, once at the start of the job; it knows to call `beginRun`, if present, at the start of each run and, likewise, `beginSubRun` at the start of each subRun.

12.6.2 About the `art::*ID` Classes

In Section 10.7.3.7 you learned about the class `art::EventID`, which describes the three-part event identifier. *art* also provides two related classes:

- `art::RunID`, a one-part identifier for a run number
- `art::SubRunID`, a two-part identifier for a subRun

The header files for these classes are found at:

```
$ART_INC/art/Persistency/Provenance/RunID.h
$ART_INC/art/Persistency/Provenance/SubRunID.h
```

Similar to the `art::Event` class discussed in Section 10.7.3.6, *art* provides `art::Run` and `art::subRun`. These contain the IDs, e.g., `art::RunID`, plus the data products for the entire run or subRun, respectively. You can find their header files at:

```
$ART_INC/art/Framework/Principal/Run.h
$ART_INC/art/Framework/Principal/SubRun.h
```

In the call to `beginSubRun` the argument is of type `art::SubRun const&`. A simplified description of this object is that it contains an `art::SubRunID` plus a collection of data products that describe the subRun. All of the comments about the class `art::Run` in the preceding few paragraphs apply to `art::SubRun`. You can find the header file for `art::SubRun` at:

```
less $ART_INC/art/Framework/Principal/SubRun.h
```

12.6.3 Use of the `override` Identifier

The `override` identifier on each of these member functions instructs the compiler to check that both the name (and spelling) of the member function and its argument list are correct; if not, the compiler will issue an error message and stop. This is a very handy feature. Without it, a misspelled function name or incorrect argument list would cause the compiler to assume that you intended to define a *new* member function unrelated to one of these optional *art*-defined member functions. This would result in a difficult-to-diagnose run-time error: *art* would simply not recognize your member function and would never call it.

Always provide the `override` identifier when using any of the optional *art*-defined member functions.

For those with some C++ background, the three member functions `beginJob`, `beginRun` and `beginSubRun` are declared as virtual in the base class, `art::EDAnalyzer`. The `override` identifier is new in C++-11 and will not be described in older text books. It instructs the compiler that this member function is intended to override a virtual function from the base class; if the compiler cannot find such a function in the base class, it will issue an error.

12.6.4 Use of `const` References

In `Optional_module.cc` the argument to the `beginRun` member function is a `const` reference to an object of type `art::Run` that holds the current run ID and the collection of data products that together describe the run. If you take a snapshot of a running *art* job you will see that, at any time, there is exactly one object of type `art::Run`. This object is

owned by *art*. *art* gives modules access to it when it (*art*) calls the modules' `beginRun` and `endRun` member functions.

Because the object is passed by reference, the `beginRun` member function does not get a copy of the object; instead it is given access to it. Because it is passed by `const` reference in this example, your analyzer module may look at information in the object but it may not add or change information to the `art::Run` object.

There is a very important habit that you need to develop as a user of *art*. Many member functions in *art*, in the Workbook code and very likely in your experiment's code, will return information by `&` or by `const&`. If you receive these by value, not by reference, then you will make copies that waste both CPU and memory; in some cases these can be significant wastes. Unfortunately there is no way to tell the compiler to catch this mistake. The only solution is your own vigilance.

To access the `art::Run` and `art::SubRun` objects through, for example, an `art::Event` named `event`, you can use

```
art::SubRun const& subRun = event.getSubRun();
```

for the subRun and

```
art::Run const& run = subRun.getRun();
```

for the run.

12.6.5 The `analyze` Member Function

In your `analyze` member function, if you have an `art::Event`, named `event`, you can access the associated run information by:

```
art::Run const& run = event.getRun();
```

You may sometimes see this written as:

```
auto const& run = event.getRun();
```

Both versions mean exactly the same thing. When a type is long and awkward to write, the `auto` identifier is very useful; however it is likely to be very confusing to beginners. When you encounter it, check the header files for the classes on the right hand side of the

assignment; from there you can learn the return type of the member function that returned the information.

12.7 Running this Exercise

Look at the file `optional.fcl`. This FHiCL file runs the module `Optional` on the the input file `inputFiles/input03_data.root`. Consult Table 9.1 and you will see that this file contains 15 events, all from run 3. It contains events 1 through 5 from each of subRuns 0, 1 and 2. With this knowledge, and the knowledge of the source file `Optional_module.cc`, you should have a clear idea of what this module will print out.

In your build directory, run the following command

```
art -c fcl/OptionalMethods/optional.fcl >& output/optional.log
```

The part of the printed output that comes from the module `Optional` is given in Listing 12.1. Is this what you expected to see? If not, understand why this module made the printout that it did. If you did not get this printout, double check that you followed the instructions carefully; if that still does not fix it, ask for help (see Section 3.4).

Listing 12.1: Output from `Optional_module.cc` with `optional.fcl`

```
1
2  Hello from Optional::constructor.
3  Hello from Optional::beginJob.
4  Hello from Optional::beginRun: run: 3
5  Hello from Optional::beginSubRun: run: 3 subRun: 0
6  Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 1
7  Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 2
8  Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 3
9  Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 4
10 Hello from Optional::analyze. Event id: run: 3 subRun: 0 event: 5
11 Hello from Optional::beginSubRun: run: 3 subRun: 1
12 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 1
13 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 2
14 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 3
15 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 4
16 Hello from Optional::analyze. Event id: run: 3 subRun: 1 event: 5
17 Hello from Optional::beginSubRun: run: 3 subRun: 2
18 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 1
19 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 2
```

```
20 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 3
21 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 4
22 Hello from Optional::analyze. Event id: run: 3 subRun: 2 event: 5
```

12.8 The Member Function `beginJob` versus the Constructor

The member function `beginJob` gets called once at the start of the job. The constructor of the each module is also called once at the start of the job. This brings up the question: What code belongs in the constructor and what code belongs in the `beginJob` member function?

A small number of things must be done in the constructor — see below. Other tasks can be done in either place but most experiments have found it useful to follow the rough guideline that you should put initializers and code related to *art* bookkeeping in the constructor and that you should put physics-related code in `beginJob`. Hopefully the meaning of this advice will become clear as you work through the Workbook. Your experiment may have additional, more specific, guidelines.

The correct place to initialize data members is in the constructor and, whenever possible, you should use the initailizer list syntax. Never defer initialization of a data member to the `beginJob` member function or later. When you encounter producer modules, you will learn about some more tasks that must be performed in the constructor. *This chapter has not yet been written.*

For those of you familiar with ROOT, we can provide an example of something physics-related. You should create histograms, ntuples and trees in one of the `begin` member functions, not in the constructor. In many cases you can create them in `beginJob` but there are cases in which you will need to defer creation until `beginRun` or `beginSubRun`. For example, conditions data is intrinsically time dependent and may not be available at `beginJob`-time. If creating a histogram requires access to conditions information you will need to create that histogram in `beginRun`, or `beginSubRun`, not in `beginJob`.

12.9 Suggested Activities

12.9.1 Add the Matching `end` Member functions

art defines the following three member functions:

```
void endJob     () override;
void endRun     ( art::Run const&    run    ) override;
void endSubRun  ( art::SubRun const& subRun ) override;
```

Go to your source window. In the file `Optional_module.cc`, add these member functions to the declaration of the class `Optional` and provide an implementation for each. In your implementation, just copy the printout created in the corresponding `begin` function and, in that printout, change the string "begin" to "end".

Then go to your build window and make sure that your current directory is your build directory. Then rebuild this module and run it:

```
buildtool art -c fcl/OptionalMethods/optional.fcl >& output/optional2.log
```

Consult Chapter 10 if you need to remember how to indentify that the build completed successfully. Compare the output from this run of *art* with that of the previous run: do you see the additional printout from the member functions that you added?

The solution to this activity is provided as the file `OptionalAnswer01_module.cc`. It is already built. You can run it with:

```
art -c fcl/OptionalMethods/optionalAnswer01.fcl >& output/optionalAnswer01.log
```

Does the output of your code match the output from this code?

12.9.2 Run on Multiple Input Files

In a single run of *art*, run your modified version of the module `Optional` on all of the three of the following input files:

```
inputFiles/input01_data.root
inputFiles/input02_data.root
inputFiles/input03_data.root
```

If you need a reminder about how to tell *art* to run on three input files in one job, consult Section 9.8.5.

Make sure that the printout from this job matches the description of the event loop found in Section 3.6.2.

12.9.3 The Option --trace

The *art* command supports a command line option named `--trace`. This creates additional printout that identifies every step in the event loop. Use this option to trace what *art* is doing when you run this exercise. For example

```
art -c fcl/OptionalMethods/optional.fcl --trace >& output/trace.log
```

You should be able to identify your printout among the printout from *art* and see that your printout appears in the expected place.

When you are getting an error from *art* and you don't understand which module is causing the problem, you can use `--trace` to narrow your search.

13 Exercise 4: A First Look at Parameter Sets

13.1 Introduction

In the previous few chapters you have used FHiCL files to configure *art* jobs. From Section 9.8 recall the definition of a FHiCL *table*: it is a group of FHiCL definitions delimited by braces { }. When *art* reads its run time configuration FHiCL file, it transforms the FHiCL file into a C++ representation; in that representation, each FHiCL table becomes an object of type `fhicl::ParameterSet`, which we refer to as a *parameter set*(γ).

Among other things, you have learned how to define a module label and its corresponding parameter set, the simplest case looking like:

```
moduleLabel : {
   module_type : ClassName
}
```

where the `moduleLabel` is an identifier that you define and `ClassName` is the name of a module class. *art* requires that the `module_type` parameter be present.

When you define a module label, you may enter additional FHiCL definitions (i.e., *parameters*) between the braces to form a larger parameter set. For example:

```
moduleLabel : {
   module_type      : ClassName
   thisParameter    : 1
   thatParameter    : 3.14159
   anotherParameter : "a string"
   arrayParameter   : [ 1, 3, 5, 7, 11] }
   nestedPSet       : {
                        a : 1
                        b : 2
                      }
}
```

This functionality allows you to write modules whose behaviour is run-time configurable. For example, if you have a reconstruction algorithm that depends on some cuts, the values of those cuts can be provided in this way.

13.2 Prerequisites

The prerequisite for this chapter is all of the material in Part I (Introduction) and the material in Part II (Workbook) up to and including Chapter 10. You can read this chapter without necessarily having read Chapter 11 or 12.

13.3 What You Will Learn

In Section 10.7.3.4 you saw that the constructor of a module is required to take an argument of type `fhicl::ParameterSet const&`.

In this chapter you will learn how to use this argument to read additional parameters in a parameter set. In particular, you will learn about the class `fhicl::ParameterSet` and after working through the exercises in this section, you should know how to:

1. read parameter values from a FHiCL file into a module

2. require that a particular parameter be present in a parameter set
3. use data members to communicate information from the constructor to other member functions of a module
4. print a parameter set
5. use the colon initializer syntax
6. provide a default value for a parameter (if the parameter is absent from a parameter set)
7. modify the precision of the printout of floating point types
8. recognize the error messages for a missing parameter or for a value that cannot be converted to the requested type

You will also learn:

1. that you should find out your experiment's policy about what sorts of parameters are allowed to have default values
2. an extra parameter automatically added by *art*, but only in parameter sets that are used to configure modules
3. the canonical forms of parameters

Finall, you will learn a small amount about C++ templates and C++ exceptions, just enough to understand the exercise.

13.4 Setting up to Run this Exercise

To run this exercise, you need to be logged in to the computer on which you ran Exercise 2 (in Chapter 10). If you are continuing on from a previous exercise, you need to keep both your source and build windows open.

If you are logging back in, follow the instructions in Section 10.5 to reestablish your source and build windows.

In your source window, cd to your source directory. Then cd to the directory for this exercise and look at its contents:

```
cd art-workbook/ParameterSets
ls
```

```
CMakeLists.txt    pset02.fcl        PSet03_module.cc
pset01.fcl        PSet02_module.cc  pset04.fcl
PSet01_module.cc  pset03.fcl        PSet04_module.cc
```

The source code for the first module you will run is `PSet01_module.cc` and the FHiCL file to run it is `pset01.fcl`. The file `CMakeLists.txt` is identical to that used by the previous two exericses. The remaining files are the source and FHiCL files for additional steps in this exercise.

In your build window, make sure that you are in your build directory. At this time you do not need to build any code because all code for the Workbook was built the first time that your ran buildtool.

13.5 The Configuration File `pset01.fcl`

The FHiCL file that you will run in this exericse is `pset01.fcl`. Look at this file in your source window. You will see that `pset01.fcl` defines a parameter set `psetTester`, shown below, that configures an analyzer module named `PSet01`.

```
analyzers: {
   psetTester : {
     module_type : PSet01
     a : "this is quoted string"
     b : 42
     c : 3.14159
     d : true
     e : [ 1, 2, 3 ]
     f : {
       a : 4
       b : 5
     }
   }
}
```

Figure 13.1: The FHiCL definition of the parameter set `psetTester` from `pset01.fcl`.

The parameter `module_type` is processed by *art*. All of the other parameters are processed by code in the module class `PSet01`. Additional definitions like these in a FHiCL file have the following properties:

1. The module specified by the `module_type` parameter defines which parameters must be present in this list, and which parameters are optional.
2. Each definition must be a legal FHiCL definition.
3. These definitions have no meaning, per se, to *art* or to FHiCL; they only have meaning to the C++ code in `PSet01_module.cc`.
4. Each definition may use the full power of FHiCL and my contain nested parameter sets to arbitrary depth.

Looking at the parameter set, it appears that the parameter `a` has a value that is a string of text, parameter `b`'s value is an integer number, parameter `c`'s is a floating point number, parameter `d`'s is one of the two possible boolean values, parameter `e`'s is an array of integers and that parameter `f`'s is a nested parameter set. You will learn in Section 13.6 that, from the point of view of the code in `PSet01_module.cc`, this intuition is cor-

rect. But there is one subtley: FHiCL itself has no notion of type and, inside FHiCL, all parameter values are just strings. The interpretation of a parameter value as a particular type is done by code inside `PSet01_module.cc`. The computer-science-speak for this is that FHiCL is a *type-free* langauge; this is in contrast to C++ which is a *strongly-typed* language.

13.6 The Source code file `PSet01_module.cc`

The source code for this exercise is found in the file `PSet01_module.cc`. The new features seen in this exercise are all in the definition of the constructor.

When *art* starts up, it reads the file `pset01.fcl` and, among many other things, copies the FHiCL table `psetTester` into an object of type `fhicl::ParameterSet`. When *art* calls the constructor of `PSet01`, it passes this `fhicl::ParameterSet` as the argument of the constructor, named `pset`. That is, the table named `psetTester` in the FHiCL file appears in the module as a parameter set named `pset`.

Let's examine the first part of the constructor; see Figure 13.2.

Recall from Section 13.5 that the object `pset` internally represents the value of each parameter as a string. If you ask that the value of a parameter be returned as a string, `pset` will simply return a copy of its internal representation of that parameter. On the other hand, if you ask that the value of a parameter be returned as any other type, then `pset` needs to do some additional work. For example, if you ask that a parameter be returned as an `int`, then `pset` must first find its internal string representation of that parameter; it must then convert that string into a temporary variable of the requested type and return the temporary variable. Therefore, when your code asks `pset` to return the value of a parameter, it must tell `pset` two things:

1. the name of the parameter
2. the type to which the string representation should be converted

The angle bracket syntax <> is the signature of a feature of C++ called *templates*(γ). *art* and FHiCL use templates in several prominent places. You do not need to fully understand templates — just how to use them when you encounter them. The following pages describe how to use templates when getting the value of a parameter from `pset`.

The type of the argument is `fhicl::ParameterSet const&`. The class `ParameterSet` is in the namespace `fhicl`; all identifiers in the namespace `fhicl` are found in the UPS package named **fhiclcpp**. You can find the header file for `ParameterSet` at $FHICLCPP_INC/fhiclcpp/ParameterSet.h.

```
tex::PSet01::PSet01(fhicl::ParameterSet const& pset )
      : art::EDAnalyzer(pset) {
```

The argument named `pset` is passed by `const` reference; i.e., the module is not allowed to modify it (due to `const`), and the module is given access to it, but not given a copy of it (due to the &, meaning "reference").

The fragment below illustrates how to extract values from `pset` and copy those values into local variables. The angle brackets, <>, indicate the use of a feature of C++ called *templates*(γ). This section of code is further discussed within the chapter text.

```
std::string a = pset.get<std::string>("a");
int b = pset.get<int> ("b");
double c = pset.get<double>("c");
bool d = pset.get<bool> ("d");
std::vector<int> e = pset.get< std::vector<int> >("e");
fhicl::ParameterSet f = pset.get<fhicl::ParameterSet>("f");

int fa = f.get<int>("a");
int fb = f.get<int>("b");
```

In most cases the `fhicl::ParameterSet` representation of a FHiCL table does NOT contain the name of the FHiCL table (`psetTester` in this case); it contains only the parameters defined between the braces. The exception is for FHiCL tables used to configure *art* modules; in this case *art* adds an extra parameter named `module_label` whose value is the the module label.

```
std::string module_type =
          pset.get<std::string>("module_type");
std::string module_label =
          pset.get<std::string>("module_label");
```

Figure 13.2: First part of constructor in `PSet01_module.cc`

When you ask for the value of a parameter, the name of the parameter is specified as a familiar function argument while the return type is specified between the angle brackets. The name between the angle brackets is called a *template argument*. If you do not supply a template argument, then your code will not compile.

For example, the line that sets the parameter `a` that reads

```
std::string a = pset.get<std::string>("a");
```

It first declares a local variable named `a` that is of type `std::string` and then asks `pset` to do the following:

1. Check if it has a parameter named `a`.
2. If it has this parameter, return it as a string.

The returned value is used to initialize the local variable, `a`. Section 13.9 will describe what happens if `pset` does not have a parameter named `a`.

It is not required that the local variable, `a`, have the same name as the FHiCL parameter `a`. But, with rare exceptions, it is a good practice to make them either exactly the same or close to the same.

The following line, that sets the parameter `b` and that reads

```
int b = pset.get<int>("b");
```

is similar to the previous line; the main difference is that `pset` will convert the string to an `int` before returning it. `pset` knows that it must perform the conversion to `int` because the template argument tells it to. Section 13.9 will describe what happens if the string cannot be converted to an `int`.

It is beyond the scope of this chapter to discuss how the template mechanism is used to trigger automatic type conversions. It is sufficient to always remember the following: when you use the `get` member function of the class `fhicl::ParameterSet`, the template argument must always match the type of the variable on the left hand side. Templates will be discussed in Section 13.8.

The authors of FHiCL could have designed a different interface, such as

```
std::string a = pset.get_as_string("a");
std::string b = pset.get_as_int   ("b");
```

Instead they chose to write it using templates. The reason for this choice is that it allows one to add new types to FHiCL without needing to recompile FHiCL. How you do this is beyond the scope of this chapter. You now know everything that you need to know about templates in order to use `fhicl::ParameterSet` effectively.

The rest of the lines in that section of code extract the remaining parameters from `pset` and make copies of them in local variables. The remainder of the constructor, shown in Figure 13.3, prints the values of these parameters; these make the printout in lines 1 through 11 of Listing 13.1.

Your code may ask for the values of parameters from a `ParameterSet` in any order, and any number of times, including zero.

Two final comments on `PSet01_module.cc`. First, the `analyze` member function is empty. Nevertheless, it must be present because *art* requires all analyzer modules to provide a member function named `analyze`. If we removed this member function from the class `PSet01`, then the module would not compile. Second, the argument of the `analyze` member function is not used; therefore it is not given a name. Were it given a name, the compiler would complain that the argument was never used. When no name is given the compiler understands that it is your intention not to use the argument. Even though the code does not use the argument, its type must be present because the number, type and order of the arguments is part of the signature of a function.

13.7 Running the Exercise

Now let's see what happens when you run the job. In your build directory, run the following command

```
art -c fcl/ParameterSets/pset01.fcl >& output/pset01.log
```

The expected output from this command is shown in Listing 13.1.

The module reads in the parameter set and then prints out each of the values in several different ways. Check that the printout matches the definitions of the parameters from `pset01.fcl`. Understand the relationship between the printout and the lines in the source file `PSet01_module.cc`.

```
std::cout << "\n----------\nPart 1:\n";
std::cout << "a : " << a << std::endl;
std::cout << "b : " << b << std::endl;
std::cout << "c : " << c << std::endl;
std::cout << "d : " << d << std::endl;

std::cout << "e :";
for ( int i: e ){
  std::cout << " " << i; }
std::cout << std::endl;
```

The following lines show one way to print the two values, `a` and `b` from the parameter set `f`.

```
std::cout << "f.a : " << fa << std::endl;
std::cout << "f.b : " << fb << std::endl;

std::cout << "module_type: " << module_type << std::endl;
std::cout << "module_label: " << module_label << std::endl;
```

The following three lines show show two other ways, using the `to_string()` and `to_indented_string()` member functions of the class `fhicl::ParameterSet`. These lines make the printout found in lines 13 through 18 of Listing 13.1.

```
std::cout << "\n----------\nPart 2:\n";
std::cout << "f as string: " << f.to_string() << std::endl;
std::cout << "f as indented-string:\n"<< f.to_indented_string() << std::endl

std::cout << "\n----------\nPart 3:\n";
std::cout << "pset:\n" << pset.to_indented_string() << std::endl;
```

The last two lines use the `to_indented_string()` member function to print everything found in the parameter set `psetTester`. These lines make the printout found in lines 21 through 36 of Listing 13.1.

Figure 13.3: Remainder of the constructor in `PSet01_module.cc`

Listing 13.1: Output from `PSet01` with `pset01.fcl` (*art*-standard output not shown)

```
--------------------
Part 1:
a : this is quoted string
b : 42
c : 3.14159
d : 1
e : 1 2 3
f.a : 4
f.b : 5
module_type:  PSet01
module_label: psetTester

--------------------
Part 2:
f as string:          a:4 b:5
f as indented-string:
a: 4
b: 5


--------------------
Part 3:
pset:
a: "this is quoted string"
b: 42
c: 3.14159
d: true
e: [ 1
   , 2
   , 3
   ]
f: { a: 4
     b: 5
   }
module_label: "psetTester"
module_type: "PSet01"
```

13.8 Member Function Templates and their Arguments

Now that you have seen templates, we can introduce some more language that you will need to know. In the above examples, `get<std::string>` and `get<int>` are *member functions* of the class `ParameterSet`.

On its own, `get` is called a *member function template*; this means that `get` is a set of rules to write a member function. The member function can only be written once the template's argument has been specified. In the future, when we refer to `get`, we will call it by its proper name:

```
ParameterSet::get<T>
```

or, sometimes, just `get<T>`. In the notation `<T>`, the angle brackets indicate that `get` is a template and the capital letter `T` is a dummy argument that indicates that if you want to use the template, you must supply one template argument. The choice of the letter `T` as the name of the dummy argument is a mnemonic for `Type`, indicating that the template argument is usually the name of a type.*

If you are familar with template meta-programming you can find the source for the class `fhcil::ParameterSet` in the files:

```
$FHICLCPP_INC/fhiclcpp/ParameterSet.h
$FHICLCPP_DIR/source/fhiclcpp/ParameterSet.cc
```

In particular, you can find the source for `ParameterSet::get<T>`.

13.8.1 Types Known to `ParameterSet::get<T>`

This section describes the different types that can be used as the template argument for `ParameterSet::get<T>`. If you use `ParameterSet::get<T>` "out of the box", it supports the following types.

- For a parameter that has a simple value, `get<T>` supports: `bool` and `std::string`; any C++ built-in integral type, such as `int`, `unsigned` or `short`; any C++ build-in floating point type, such as `float` or `double`;

* Much later in the workbook, you will see one case in which it is something other than the name of a type.

- For a parameter whose value is another parameter set, `T` must be `fhicl::ParameterSet`.
- For a parameter with a value that is a sequence of items, all items in the sequence must be of the same type and `get<T>` allows `T` to be `std::vector<S>`, where the template argument `S` is any of the types given in the previous two bullets.

13.8.2 User Defined Types

This sub-section on reading user defined types from a parameter set is for experts only. 

You can write helper functions that will allow the type `T` to be almost any type that you might want. How to do this is beyond the scope of this chapter. For an example see the files:

```
$TOYEXPERIMENT_DIR/source/toyExperiment/Utilities/ParameterSetHelpers.h
$TOYEXPERIMENT_DIR/source/toyExperiment/Utilities/ParameterSetHelpers.cc
```

These files allow you to define a FHiCL parameter like:

```
zaxis : [  0., 0., 1.]
```

and to read it as

```
auto zaxis = pset.get<CLHEP::Hep3Vector>("zaxis");
```

13.9 Exceptions

13.9.1 Error Conditions

There are two sorts of error conditions that may occur when reading parameters from a parameter set:

1. The requested parameter is not present in the parameter set.
2. The requested parameter is present but cannot be converted into the requested type.

To give an example of the second sort, suppose that on line 6 of Listing 13.1 you change the FHiCL definition of the parameter `c` from `3.14159` to “test": Now consider what

happens when you try to read this parameter as a double, as is done on the line

```
double c = pset.get<double>("c");
```

from the listing in Figure 13.2. The code will correctly find that parameter `c` exists but it will produce an error when it tries to convert the string “test" to a `double`.

In both of these cases, the code inside `pset` will tell *art* to stop processing events and to perform an orderly shutdown, which will be described in the next sub-section.

13.9.2 Error Handling

From time to time code within *art* will discover that, because of some error condition, it cannot continue to process events. When this happens *art* can be configured to stop processing events and then to do one of several different things:

1. It can attempt an orderly shutdown, described below.
2. It can write the offending event to a separate output file and continue normal with the next event.
3. It can skip the module in which the problem occured and continue normal processing with the next module.
4. There are several other options that cannot be described here because the necessary background information has not yet been established.

When *art* attempts an orderly shutdwon, it will:

1. Write a message to the log file that describes what happened.
2. Record the error condition that stopped processing; this information will be written to all output event-data files.
3. Call the endSubRun member function of every module.
4. Call the endRun member function of every module.
5. Call the endJob member function of every module.
6. Properly flush and close all output and log files.

7. Perform a few other clean up and shutdown actions for parts of *art* that have not yet been discussed.

8. Return a non-zero status code to the parent process; the status code is the number that appears on the last line of your *art* output, the line that begins "Art has completed ...".

For most sorts of errors, the orderly shutdown will be successful and your work up to the error will be preserved.

But there are circumstances for which the orderly shutdown will fail. One example of this is if you have reached your disk quota and there is no disk space to hold more output.

For all cases but one, *art*'s default behaviour is to attempt an orderly shutdown. The one non-standard case is when *art* is unable to find a requested data product; in that case the default behaviour is to continue with the next module.

These default behaviours can be changed by adding lines the the FHiCL file. A discussion of these features is beyond the scope of this chapter. When the section that describes how to do this is written, a link to that section will be added here.

.

The technology that *art* uses to interupt event processing and to take one of the possible follow-on actions is a feature of C++ called exceptions. When *art* stops event processing it is said to *throw an exception*; this phrase will be used throughout the Workbook. The topic of exceptions is much to complex to even sketch in this chapter; at this time you only need to understand that the phrase *throw an exception* means to stop event processing and to take one of the allowed follow-on actions, usually an orderly shutdown. A chapter yet to be written will describe how to use exceptions in your own code to tell *art* to interupt processing.

.

13.9.3 Suggested Exercises

In `pset01.fcl`, remove the definition of the parameter `b`. Rerun *art*. You should see an error message like that shown in Listing 13.2. Read the error message and understand

what it is telling you so that you will recognize the error message if you make this mistake in the future.

Listing 13.2: Output from `PSet01` with `pset01.fcl` (parameter `b` removed)

```
%MSG-s ArtException:  PSet01:psetTester@Construction 14-Jul-2013 19:38:19 CDT
ModuleConstruction
cet::exception caught in art
---- Can't find key BEGIN
  b
---- Can't find key END
%MSG
Art has completed and will exit with status 8001.
```

Note too that the completion status is non-zero.

In `pset01.fcl`, restore the definition of `b` and change the definition of `c` to `"test"` . Rerun *art*. You should see an error message like that shown in Listing 13.3. Again, read the error message and understand it.

Listing 13.3: Output from `PSet01` with `pset01.fcl` (parameter `c` misdefined)

```
%MSG-s ArtException:  PSet01:psetTester@Construction 14-Jul-2013 19:42:54 CDT
ModuleConstruction
cet::exception caught in art
---- Type mismatch BEGIN
  c
  ---- Type mismatch BEGIN
    error in float string:
    test
    at or before:
  ---- Type mismatch END
---- Type mismatch END
%MSG
Art has completed and will exit with status 8001.
```

13.10 Parameters and Data Members

Very often information from the parameter set is needed in a member function of the module class. The way to propagate this information from the parameter set to the member function is to store the values of these parameters as data members of the module class. This is illustrated in the two files `PSet02_module.cc` and `pset02.fcl`. Open these files with an editor and follow along with the description below.

If you need to refamiliarize yourself with the concept of data members of a class, refer to Section 6.6.2.

There are three things to notice in `PSet02_module.cc`.

1. The class declares three data members named `b_`, `c_`, and `f_`. These are declared in the private section so that only the module itself can see them.
2. In the constructor, these three data members are initialized to values extracted from the module's parameter set.
3. In the analyze member funcion all three data members are printed out.

If you need to refamiliarize yourself with the colon intializer syntax, refer to Section 6.6.5. If you need to refamiliarize yourself with the conventions about underscore characters in the names of data members refer to Section 6.6.7.2.

To run this example, enter

```
art -c fcl/ParameterSets/pset02.fcl >& output/pset02.log
```

The expected output from this is given in Listing 13.4.

Listing 13.4: Output from `PSet02` with `pset02.fcl`

```
1 Event number: run: 1 subRun: 0 event: 1  b: 42  c: 3.14159  f: a:4 b:5
2 Event number: run: 1 subRun: 0 event: 2  b: 42  c: 3.14159  f: a:4 b:5
3 Event number: run: 1 subRun: 0 event: 3  b: 42  c: 3.14159  f: a:4 b:5
```

This example is only relevant when parameters are actually used in member functions. If a parameter is used only inside the constructor, do not store it as a data member; instead you should store it as a local variable of the constructor. This brings up a "best practice:" always declare a variable in the narrowest scope that works.

13.11 Optional Parameters with Default Values

It is sometimes convenient to provide a default value for a parameter. Default values may be provided in the source code that reads the parameter set. This mechanism is illustrated by the files `PSet03_module.cc` and `pset03.fcl`. Open these files with an editor and follow along with the description below.

You have already seen that the member function template `ParameterSet::get<T>` takes one function argument, the name of the parameter. For example,

```
int b = pset.get<int>("b");
```

It also takes an optional second function argument, a default value for the parameter. For example,

```
int b = pset.get<int>("b",0);
```

If the second argument is present, there two cases:

1. If the parameter is not defined in the FHiCL file, then the second argument is returned as the value of the call to `get`.
2. If the parameter is defined in the FHiCL file, then the second argument is ignored and the value read from the FHiCL file is returned as the value of the call to `get`.

When reading the code in this example you will encounter the expression:

```
std::vector<double>(5,1.0);
```

This tells the compiler to instantiate an object of type `std::vector<double>`, set its size to 5 and initialize elements 0 through 4 to have the value 1.0. If you are not familiar with this syntax, you can read about it in the documentation for the C++ Standard Library (see Section 6.7).

This expression appears as the second argument of the second call to the member function `pset.get<T>`. Therefore the compiler will create an unnamed temporary object (the vector of doubles) and pass that object to the member function `get<std::vector<double>>` as its the second argument; the compiler ensures that, once function call has completed, the temporary object is deleted.

With the above explanations, the source code for this example should be reasonably self-explanatory; it looks for two parameters named `debugLevel` and `g` and supplies default values for each of them. Look at the file `pset03.fcl`; you will see that the parameters `debugLevel` and `g` are not present in the `testPSet` parameter set; therefore printout will show the default values.

To run this example,

```
art -c fcl/ParameterSets/pset03.fcl >& output/pset03.log
```

Listing 13.5: Parameter-related portion of output from `PSet03` with `pset03.fcl`

```
1 debug level: 0
2 g:  1 1 1 1 1
```

The expected output from this is given in Listing 13.5.

As a suggested exercise, edit `pset03.fcl` and, in the parameter set `testPSet`, provide definitions for the parameters `debugLevel` and `g`. Make their values different from the default values. Rerun *art* and verify that the module has correctly read in and printed out the values you defined.

13.11.1 Policies About Optional Parameters

Allowing *optional* parameters is important for developing, debugging and testing; if all parameters were required all of the time, the complete list of parameters could become unwieldy[†]. On the other hand, the use of optional parameters can make it difficult to audit the physics content of a job. Therefore experiments typically have policies for what sorts of parameters may have defaults and what sorts may not. For example, your experiment may prohibit default values for parameters that define the physics behavior, but allow them for parameters that control printout and other diagnostics.

Consult your experiment to learn what policies you should follow.

13.12 Numerical Types, Precision and Canonical Forms

FHiCL recognizes numbers in both fixed point and exponential notation, for example `123.4` and `1.234e2`; the letter `e` that separates the exponent can be written in either upper or lower case.

In the preceding exercises you defined some numerical values in a FHiCL file, read them into your code and printed them out; the printed values exactly matched the input values. The values used in those exercises were carefully chosen to avoid a few surprises: there

[†]FHiCL has several features that make it easier to deal with large parameter sets. *This will be explained in a future chapter.*

Table 13.1: Canonical forms of numerical values in FHiCL files

Number	Canonical Form	Number	Canonoical Form
2	2	1.234E2	1.234e2
2.	2	1.23456E5	123456
2.0	2	1.23456E6	1.23456e6
2.1E2	210	1234567	1.234567e6
+210	210	0.01	1E-2

are cases in which the printed value will be an equivalent, but not identical, form. This section discusses those cases and provides some examples.

When FHiCL recognizes that a parameter value is a number it converts the number into a *canonical form* and stores the canonical form as a string. The transformation to the canonical form preserves the full precision of the number and involves the following steps:

1. The canonical form has no insignificant characters:
   (a) no insignificant trailing zeros
   (b) no insignificant trailing decimal point
   (c) no insignficant leading plus sign
   (d) no insignficant leading plus sign in the exponent
2. If a number is specified in exponential notation and if the number can be represented as a integer without loss of precision, and if the resulting integer has 6 or fewer digits, then the canonical form is the integer. For example, the canonical form of `1.23456E5` is `123456` but the canonical form of `1.23456E6` is `1.23456e6`.
3. The canonical form of all other floating point numbers is exponential notation with a single, non-zero digit to the left of the decimal point.
4. The canonical form of all strings includes beginning and ending quotes; this is true even if the string contains no embedded whitespace or other special characters.

Some examples of numbers and their canonical forms are given in Table 13.1.

If a numerical value, when expressed as a fixed point number, has no fractional part, your code may ask for the parameter to be returned as either a floating point type (

Listing 13.6: Output from `PSet04` with `pset04.fcl`

```
1 parameter a as a string: 1.23456e6
2 parameter a as a double: 1.23456e+06
3 parameter a as int:      1234560
4 parameter b as a string: 3.1415926
5 parameter b as a double: 3.14159
6 parameter b as a double with more significant figures: 3.1415926
7 parameter c as a string: 1
8 parameter c as an int:   1
```

such as `double` or `float`) or as an integral type (such as `int`, `short unsigned` or `std::size_t`). For example, fourth non-blank line in the listing in Figure 13.2 was written

```
int b = pset.get<int>("b");
```

It might also have been written

```
double b = pset.get<double>("b");
```

which would do the expected thing: given the input from `pset01.fcl`, it would read the value `42` into a variable of type double.

On the other hand, if a numercial value, when expressed as a fixed point number, does have a fractional part, you may only ask for the parameter to be returned as a floating point type. If you ask for such a value as an `int`, the `ParameterSet::get<int>` member function will throw an exception; similarly for all other integral types. This behavior may not be intuitive: the authors of *art* could have decided, instead, to discard the fractional part and return the integer part. They chose not to do this because when this situations occurs, it is almost always an error.

13.12.1 Suggested Exercises

The above ideas are illustrated by the files `PSet04_module.cc` and `pset04.fcl`. To run this example,

```
art -c fcl/ParameterSets/pset04.fcl >& output/pset04.log
```

The expected output from this is given in Listing 13.6. Read the source code and the

Listing 13.7: Output from `PSet04` with modified `pset04.fcl` (intentional error)

```
1 %MSG-s ArtException:  PSet04:pset@Construction 28-Jul-2013 23:39:31 CDT
  ModuleConstruction
2 cet::exception caught in art
3 ---- Type mismatch BEGIN
4   c
5   narrowing conversion
6 ---- Type mismatch END
7 %MSG
8 Art has completed and will exit with status 8001.
```

FHiCL file; then examine the output. The first three lines show three different printed formats of the parameter `a`, with the first being the canonical form. While all forms are equal to the number found in the FHiCL file, they all have different formats. Understand why each line has the format it does.

Line 4 shows the canonical form of the parameter `b`. Line 5 shows what is printed using the default C++ settings; the two least significant characters were dropped. The code that produces line 6 shows how the use the `precision` function from the C++ Standard Library to tell C++ to print more significant figures.

If you modify the precision of `cout`, it will change the format of the printout for the rest of the job; usually this is a bad thing. To avoid this, `PSet04_module.cc` illustrates how to save and restore the precision of `cout`.

Line 7 shows the canonical form of the parameter `c` and line 8 shows the default C++ printed form of the integer.

For the next exercise, edit `pset04.fcl` and change the value of `c` to something with a fractional part. Rerun *art*; you should see that it throws an exception because it is illegal to read a numeric value with a fractional part into a variable of integral type. The error message from *art* is shown in Listing 13.7. Read the error message and understand what it is telling you so that you will recognize the error message if you make this mistake yourself.

14 Exercise 5: Making Multiple Instances of a Module

14.1 Introduction

In a typical HEP experiment is often necessary to repeat one analysis several times, with each version differing only in the values of some cuts; this is frequently done to tune cuts or to study systematic errors. Very often it is both convenient and efficient to run all of the variants of the analysis in a single job.

A powerful feature of *art* is that it permits you to run an *art* job in which you define and run many instances of the same module; when you do this, each instance of the module gets its own parameter set. In this chapter you will learn how to use this feature of *art*.

14.2 Prerequisites

The prerequisite for this chapter is all of the material in Part I (Introduction) and the material in Part II (Workbook) up to and including Chapter 13, but excluding Chapter 12.

14.3 What You Will Learn

In this chapter you will learn how to run an *art* job in which you run the same module more than once. This exercise will make it clear why *art* needs to distinguish the two ideas of *module label* and `module_type`.

14.4 Setting up to Run this Exercise

To run this exercise, you need to be logged in to the computer on which you ran Exercise 2 (in Chapter 10). If you are continuing on from a previous exercise, you need to keep both your source and build windows open.

If you are logging back in, follow the instructions in Section 10.5 to reestablish your source and build windows.

In your source window, `cd` to your source directory. Then `cd` to the directory for this exercise and look at its contents

```
cd art-workbook/ModuleInstances
ls
```

```
CMakeLists.txt  magic.fcl  MagicNumber_module.cc
```

The source code for the first module you will run is `MagicNumber_module.cc` and the FHiCL file to run it is `magic.fcl`. The file `CMakeLists.txt` is identical that used by the previous two exericses.

In your build window, make sure that you are in your build directory. At this time you do not need to build any code becaue all code for the Workbook was built the first time that your ran `buildtool`.

14.5 The Source File `Magic_module.cc`

The source code for this exercise is found in the file `Magic_module.cc`. Look at this file and you should see the following features, all of which you have seen before.

1. The file declares and defines a class named `MagicNumber` that follows the rules to be an *art* analyer module.
2. The class has a constructor and an `analyze` method.
3. The class has a data member named `magicNumber_`, of type `int`.
4. The class initializes `magicNumber_` by reading a value from its parameter set; the name of the parameter is `magicNumber` (without the underscore).

5. The parameter `magicNumber` is a required parameter.
6. Both the constructor and the `analyze` method print an informational message that includes the value of `magicNumber_`.

14.6 The FHiCL File `magic.fcl`

The FHiCL file used to run this exercise is `magic.fcl`. Look at this file and you should see the following features:

1. Compared to previous exercises, The FHiCL names `process_name`, `source` and `services` have no important differences.
2. In the `analyzers` parameter set, inside the `physics` parameter set, you will see the definition of four module labels, `boomboom`, `rocket`, `flower` and `bigbird`*. The value of each definition is a parameter set.
3. The first three of these parameter sets tell *art* to run the module `MagicNumber` and each provides a value for the required `magicNumber` parameter†
4. The last parameter set tells *art* to run the module `First`, the source for which was discussed in Chapter 10; this module does not need any additional parameters.
5. The path `e1` contains the names of all of the module labels from the `analyzers` parameter set.

14.7 Running the Exercise

In your build directory, run the following command

```
art -c fcl/ModuleInstances/magic.fcl >& output/magic.log
```

The expected output from this command is shown in Listing 14.1; for clarity, the printout made by *art* has been elided. Compare this printout to the printout from your run; it should

* All of these are nicknames of ice hockey players who played for the Montreal Canadiens ice hockey team; all of them have had their sweater number retired

†In each case the magic number is the sweater number of the hockey player whose nickname is the module label.

Listing 14.1: Output using `magic.fcl`

```
1  MagicNumber::constructor:  magic number: 9
2  MagicNumber::constructor:  magic number: 5
3  Hello from First::constructor.
4  MagicNumber::constructor:  magic number: 10
5  MagicNumber::analyze: event: run: 1 subRun: 0 event: 1 magic number: 9
6  MagicNumber::analyze: event: run: 1 subRun: 0 event: 1 magic number: 5
7  Hello from First::analyze. Event id: run: 1 subRun: 0 event: 1
8  MagicNumber::analyze: event: run: 1 subRun: 0 event: 1 magic number: 10
9  MagicNumber::analyze: event: run: 1 subRun: 0 event: 2 magic number: 9
10 MagicNumber::analyze: event: run: 1 subRun: 0 event: 2 magic number: 5
11 Hello from First::analyze. Event id: run: 1 subRun: 0 event: 2
12 MagicNumber::analyze: event: run: 1 subRun: 0 event: 2 magic number: 10
13 MagicNumber::analyze: event: run: 1 subRun: 0 event: 3 magic number: 9
14 MagicNumber::analyze: event: run: 1 subRun: 0 event: 3 magic number: 5
15 Hello from First::analyze. Event id: run: 1 subRun: 0 event: 3
```

be exactly the same. Inspect the printout and the files `MagicNumber_module.cc` and `../FirstModule/First_module.cc`; understand why the printout is what it is.

14.8 Discussion

14.8.1 Order of Analyzer Modules is not Important

As it happens, *art* runs the four analyzer modules in the order specified in the path definition `e1`. But you must not count on this behaviour! Two of the design rules of *art* are:

1. Modules may only communicate with each other by putting information into, and reading information from, the `art::Event`.
2. Analyer modules may not put information into the `art::Event`.

Therefore *art* is free to run analyzer modules in any order.

For producer modules, which may add information to the event, the order of execution is often very important. When you reach the exercises that run producer modules, you will be told how to specify the order of execution.

You may wish to review some of the other ideas about *art* paths that are described in Section 9.8.8.

14.8.2 Two Meanings of *Module Label*

In the preceeding discussion, the name *module label* was used in two subtly different ways, as is illustrated by the module label `rocket`:

1. `rocket` identifies a parameter set that is used to configure an instance of the module `MagicNumber`.
2. `rocket` is also used as the name of the module instance that is configured using this parameter set; the elements in the path `e1` are all the names of module instances.

Clearly these two meanings are very closely related, which is why the same name, *module label*, is used for both ideas. Throughout the remainder of this document suite the name *module label* will be used for both meanings; the authors believe it will be clear from the context which meaning is intended. This is standard usage within the *art* community.

14.9 Suggested Exercise

Edit `magic.fcl` and do the following:

1. Add a new analyzer module label that configures an instance of the module `Optional` from Chapter 12.
2. Add the new module label to `e1`.

Then re-run `magic.fcl`. Do you see the expected additional printout?

14.10 Review

After working through this exercise, you should:

1. Know how to run multiple instances of the same module within one *art* job.
2. Understand that *art* does not guarantee the order in which analyzer modules will be run.

3. Understand the two senses in which the name *module label* is used: as the name of a parameter set and as the name of the corresponding instance of a module.

15 Exercise 6: Accessing Data Products

15.1 Introduction

Section 10.7.3.6 described the class `art::Event` as an `art::EventID` plus a collection of data products. The concept of a data product was described in Section 3.6.4. You have already done several exercises that made use of the `art::EventID` and in this chapter you will do your first exercises that use a data product.

15.2 Prerequisites

Prerequisites for this chapter include all of the material in Part I (Introduction) and the material in Part II (Workbook) up to and including Chapter 13.

You must also be familiar with the toy experiment described in Section 3.7.

This exercise will use class templates and member function templates in several places. The use of templates was introduced in Section 13.6. Recall that a class template is a set of rules for creating a class and that a member function template is a set of rules for creating a member function. You need to know how to use templates but you do not need to know how to write one. You will need a minimal understanding of the class template `std::vector`, which is part of the C++ Standard Library. If you understand the following four points, then you understand enough about `std::vector` for this exercise. If `t` is an object of type `std::vector<T>`, then:

1. `t` behaves much like an array of objects of type `T`. The main difference is that capacity of the array automatically grows to be large enough to hold all of the elements in the array.

2. The identifier inside the angle brackets is called a *template argument* and it is usually the name of a C++ type. *
3. The dynamic sizing occurs in the middle of a running program; not at compile time.
4. This expression sets `nEntries` to the number of entries in `t`:
   `std::size_t nEntries = t.size();`

.

15.3 What You Will Learn

In this exercise you will learn about:

1. the data type `tex::GenParticleCollection`
2. the four-part name of an *art* data product
3. the class `art::InputTag`
4. the class template `art::Handle`
5. the class template `art::ValidHandle`
6. the member function templates of `art::Event`:
   - `getByLabel( art::InputTag, art::Handle<T>) const;`
   - `getValidHandle<T>( art::InputTag ) const;`

15.4 Background Information for this Exercise

The input files used for the *art* workbook contain data products created by a workflow that simulates the response of the toy detector to a generated event, described in Section 3.7.2. The first step in this workflow is to use an event generator to create a collection of generated particles, which is stored in the `art::Event` as a data product. That is, there is

*You will only see one case in the entire workbook in which it is something other than a the name of a C++ type; and this will be during a short side trip to discuss the **cetlib** utility library.

a single data product that holds a collection of generated particles; there is not one data product per generated particle.

In this exercise you will retrieve this data product and print the number of generated particles in each event. *A future chapter will look at the properties of individual generated particles.*

15.4.1 The Data Type `GenParticleCollection`

Each generated particle in the simulated event is described by an object of type `tex::GenParticle`. All of the generated particles in a given event are stored in an object of type `tex::GenParticleCollection`. This object is written to the `art::Event` as a data product.

The header files that describe these two classes, `GenParticle.h` and `GenParticleCollection.h`, are found under:
`$TOYEXPERIMENT_INC/toyExperiment/MCDataProducts/`

The content of `GenParticleCollection.h` is shown in Listing 15.1; the code guards and comments have been omitted. This header uses a typedef to declare that the name `tex::GenParticleCollection`; is a synonym for `std::vector<tex::GenParticle>`.

Listing 15.1: Contents of `GenParticleCollection.h`

```
1
2 #include "toyExperiment/MCDataProducts/GenParticle.h"
3
4 #include <vector>
5
6 namespace tex {
7
8   typedef std::vector<GenParticle> GenParticleCollection;
9 }
```

Why did the authors of the workbook decide to use a typedef and not simply ask you to code `std::vector<GenParticle>` when needed? The reason is future-proofing. Suppose that down the road the authors find that they need to change the definition of `tex::GenParticleCollection`; if you used the typedef, it is much more likely

that your code will continue to compile and work correctly as is. If, on the other hand, you used `std::vector<GenParticle>`, then you would need to identify and edit every instance.

Please use the typedef `GenParticleCollection` in your own code and do not hand-substitute its definition.

Why did the authors of the workbook decide to call this typedef `GenParticleCollection` and not, for example, `GenParticleVector`? The answer is a different sort of future-proofing. The C++ standard library provides class templates other than vectors that are collections of objects, and one can imagine a scenario in which it would make sense to change `GenParticleCollection` to use a collection type, such as `std::deque` for example. In such a scenario, the following definition would make perfect sense to the C++ compiler but would be misleading to human readers:
`typedef std::deque<GenParticle> GenParticleVector`
The generic name *Collection* avoids this problem.

15.4.2 Data Product Names

Each *art* data product has a name that is a text string with four fields, delimited by underscore characters (_) that represent, in order, the data type, module label, instance name and process name, e.g.,:

`MyDataType_MyModuleLabel_MyInstanceName_MyProcessName`

Each data product name must be unique within an *art* event-data file. The fields in the data product name may only contain the following characters[†]:

- `a...z`
- `A...Z`
- `0...9`
- `::` (double colon)

[†]Experts may want to know that in an *art* event-data file, each data product is stored as a `TBranch` of a `TTree` named `Event`. Only these characters are legal in a `TBranch` name. The name of the `TBranch` is the name of the data product, hence the restriction.

In particular, periods, dashes, commas, underscores, semicolons, white space and single colons are not allowed; underscores are only allowed as the field separator, not within a field.

About each field:

1. The data type field is the so-called *friendly name* name of the data type for the data product; friendly names are discussed below.
2. The module label field is the label of the module that created the data product. Note that it is the module *label* as specified in the FHiCL file, not the `module_type`.
3. A given module instance in a given *art* process may make many data products of the same type. These are distinguished by giving each a unique instance name. An empty string is a valid instance name and in fact is the default. The other three fields must be non-empty strings.
4. The process name field holds the value of the `process_name` parameter from the FHiCL file for the *art* job that created the data product.

The friendly name of a data type is a concept that *art* inherited from the CMS software suite. You will never need to write friendly names but you will need to recognize them. Knowing the following rules will be sufficient in most cases:

1. If a type is not a collection type, then its friendly name is the fully qualified name of the class.
2. If a type is `std::vector<T>`, its friendly name is `Ts`; the mnemonic is that adding the letter "s" makes it plural.
3. If a type is `std::vector<std::vector<T> >`, its friendly name is `Tss`. And so on.
4. If a type is `cet::map_vector<T>`, its friendly name is `Tmv`.

The full set of rules is given in the Users' Guide.

Corollaries of the above discussion include:

- None of the four fields in a product name may contain an underscore character: otherwise the parsing of the name into its four fields is ambiguous.

- If an *art* event-data file is populated by running several *art* jobs, each of which adds some data products, then each *art* job in the sequence must have a unique `process_name`.

15.4.3 Specifying a Data Product

To identify a data product, *art* requires that you specify the data type, module label and instance name fields (an empty string is a valid instance name). If the event contains exactly one data product that matches this specification, then *art* allows a wild card match on the process name field. If the event contains more than one data product that matches this specification, then *art* requires that you also specify the process name. I.e., *art* allows a wild card match only on the process name field, not on the others.

To tell *art* which data type you want, you use a template argument. To specify the other three fields, module label, instance name and process name, you use an object of type `art::InputTag`. Why is the data-type field treated differently than the others? This method allows *art* to look after the translation of the data type to its friendly name. Users of *art* never need to learn how to do this translation.

The header for `art::InputTag` is found in the file
`$ART_INC/art/Utilities/InputTag.h`.
You can construct an input tag by passing it a string with the three fields separated by colons, e.g.,:

```
art::InputTag tag("MyModuleLabel:MyInstanceName:MyProcessName");
```

For this exercise, the full specification of the input tag includes only the module label and the process name:

```
art::InputTag tag("evtgen::exampleInput");
```

The double colon indicates that the instance name (which would come between the colons) is an empty string. The process name rarely needs to be specified, and in fact it is not needed in this exercise. It will be sufficient to specify the input tag as

```
art::InputTag tag("evtgen");
```

There are other constructors for `art::InputTag` and there are accessor methods that provide access to the individual fields. You can learn about these by looking at the header

file but you will not use these features in this exercise.

15.4.4 The Data Product used in this Exercise

The input files used for this exercise contain data products, one of which this exercise will use. This data product has the following attributes:

- it has a data type of `tex::GenParticleCollection`
- it is produced by a module with the label `evtgen`
- its instance name is an empty string
- it is produced by an *art* job with the process name `exampleInput`.

15.5 Setting up to Run this Exercise

To run this exercise, you need to be logged in to the computer on which you ran Exercise 2 (in Chapter 10).

If you are continuing on from a previous exercise, you need to keep both your source and build windows open.

If you are logging back in, follow the instructions in Section 10.5 to reestablish your source and build windows.

In your source window, cd to your source directory. Then cd to the directory for this exercise and look at its contents:

```
cd art-workbook/ReadGenParticles
ls
CMakeLists.txt
readGens1.fcl     ReadGens1_module.cc
readGens2.fcl     ReadGens2_module.cc
readGens3.fcl     ReadGens3_module.cc
```

In this exercise you will run three modules that differ in only a few lines. The three source files use different syntax to accomplish the same thing. Most of the subsequent exercises in the workbook will use the syntax shown in the third version, `ReadGens3_module.cc`, and we recommend using this syntax in most cases. A description of the first two here serves as a pedagodical progression. You will likely see all three types of syntax in your experiment's code.

15.6 Running the Exercise

You will run the exercise from your build directory in your build window. The code is already built. To run this exercise, cd to your build directory and type the command:

```
art -c fcl/ReadGenParticles/readGens1.fcl >& output/readGens1.log
```

This will make the usual *art* output, interspersed with the output made by `readGens1.fcl`. The output from this module is shown in Listing 15.2. For each event it prints the event number and the number of `GenParticles` in that event.

Listing 15.2: Output using `readGens1.fcl`

```
ReadGens1::analyze event: 1 GenParticles: 7
ReadGens1::analyze event: 2 GenParticles: 3
ReadGens1::analyze event: 3 GenParticles: 3
ReadGens1::analyze event: 4 GenParticles: 3
ReadGens1::analyze event: 5 GenParticles: 5
```

15.7 Understanding the First Version, `ReadGens1`

15.7.1 The Source File `ReadGens1_module.cc`

The module `ReadGens1_module.cc` contains a new include statement for the `GenParticleCollection.h` header file. Here is the set of include statements at the top of the file:

Listing 15.3: Include statements in `ReadGens1_module.cc`

```
1 #include "toyExperiment/MCDataProducts/GenParticleCollection.h"
2
3 #include "art/Framework/Core/EDAnalyzer.h"
```

```
4  #include "art/Framework/Core/ModuleMacros.h"
5  #include "art/Framework/Principal/Event.h"
6
7  #include <iostream>
8  #include <string>
```

In the next portion of the file, notice the new data member `gensTag_` on line 16, which is initialized in the constructor using a string value that is taken from the parameter set:

```
9    class ReadGens1 : public art::EDAnalyzer {
10
11     public:
12     explicit ReadGens1(fhicl::ParameterSet const& );
13     void analyze(art::Event const& event) override;
14
15     private:
16     art::InputTag gensTag_;
17   };
18 }
```

Notice two things in the remainder of the file, below: Lines 25-26 introduce the concept of a *handle*(γ), setting `gens` as a handle to the requested `GenParticleCollection`. Lines 28-30 print out the number of entries in the data product — the same number as the number of generated particles in the event.

```
19 tex::ReadGens1::ReadGens1(fhicl::ParameterSet const& pset ):
20   art::EDAnalyzer(pset),
21   gensTag_(pset.get<std::string>("genParticlesInputTag")){
22 }
23 void tex::ReadGens1::analyze(art::Event const& event ){
24
25   art::Handle<GenParticleCollection> gens;
26   event.getByLabel(gensTag_,gens);
27
28   std::cout << "ReadGens1::analyze␣event:␣" << event.id().event()
29             << "␣GenParticles:␣"            << gens->size()
30             << std::endl;
31 }
32
33 DEFINE_ART_MODULE(tex::ReadGens1)
```

As you work through the *art* workbook you will encounter several types of handles. All of the handle types behave like pointers with additional features:

1. They have safety features that make it impossible for your code to look at a pointee that is either not valid or not available.
2. They may also have an interface that lets you access metadata that describes the pointee.

The handle is an example of a broader idea sometimes called a *safe pointer*(γ) and sometimes called a *smart pointer*(γ).

The header for the class template `art::Handle` is found in the file `$ART_INC/art/Framework/Principal/Handle.h`. This file is automatically included by the include for `Event.h`.

The `art::Handle` line tells the compiler to default construct an object of type: `art::Handle<GenParticleCollection>`. The name of the default-constructed object is `gens`. A default-constructed handle does not point at anything and, if you try to use it as a pointer, it will throw an exception. A handle in this state is said to be invalid.

The following line calls `getByLabel`, which uses its first argument (`gensTag_`) to learn three of the four elements of the name of the requested data product. It can deduce the fourth element, the data type, from the *type* of its second argument (`gens`): that is, it knows that it must look for a data product of type `tex::GenParticleCollection`. *art* has tools to compute the friendly name from the full class name, which is why you will never need to write a friendly name.

When this line is executed, the event object looks to see if it contains a data product that matches the request. There are three possible outcomes:

1. the event contains exactly one product that matches
2. the event contains no product that matches
3. the event contains more than one product that matches

In the first case, the event object will give the handle a pointer to the requested `tex::GenParticleCollection`; the handle `gens` can then be used as a pointer, as is done in the second line of the `std:cout` section. When the handle has received the pointer, it is said to be in a valid state. In the second and third cases, the event object will leave the handle in its default-constructed state and, if you try to use it as a pointer, it will

throw an exception.

If the event object finds exactly one match, it will also add two pieces of metadata to the handle. One is a pointer to an object of type `art::Provenance`, which contains information about the processing history of the data product. The second is an object of type `art::ProductID`; this is essentially a synonym for the four-field string form of the product name. Both of these will be illustrated in future exercises. .

The third case bears one more comment: the developers of *art* made a careful decision that, except for the process name field, `getByLabel` will not have a notion of "best match". When you use `getByLabel` you must unambigously specify the data product you want or *art* will leave the handle in its default-constructed state.

If the `getByLabel` member function does not find the requested data product, e.g., if you run it on a different input file or if you misspell any of the fields in the input tag, the handle will be left in its default-constructed state. In this case, the `gens->size()` call will know that the handle is invalid and will throw an exception.

In all cases but one, *art*'s response to an exception is to attempt a graceful shutdown. The one unusual case is *ProductNotFound*, which is the exception thrown by an invalid handle when you try to use it as a pointer. In this case *art* will print a warning message, skip this module and attempt to run the remaining modules in the trigger paths and end paths.

It is possible to test the state of `gens` by using the member function `gens.isValid()`, which returns a `bool`. This not illusrated in the example because in most cases we recommend that you let *art* deal with this for you.

In the preceding discussion we did not mention that `getByLabel` is actually a *member function template*. There is no explicit template argument in the `event.getByLabel` line because the C++ template mechanism is able to deduce the template argument from the type of the second argument.

The `art::Event` object supports several other ways to request data products from the event, including a way to get handles to all data products that match a partial specification. This material is beyond the scope of this exercise. .

15.7.2 Adding a Link Library to `CMakeLists.txt`

`ReadGens1_module.so` requires linking to a dynamic library that was not needed by previous exercises, namely `$TOYEXPERIMENT_LIB/libtoyExperiment_MCDataProducts.so`. This library contains the object code for the classes and functions defined in the `MCDataProducts` subdirectory of the toyExperiment UPS product. In particular it contains object code needed by the data product `tex::GenParticleCollection`.

Adding this library to the link list required a one-line modification to `CMakeLists.txt`. If you compare this file to the corresponding file for the previous exercise, you will see that `CMakeLists.txt` for this exercise contains one additional line:

```
${TOYEXPERIMENT_MCDATAPRODUCTS}
```

The string `TOYEXPERIMENT_MCDATAPRODUCTS` is a cmake variable that was defined when you first ran the buildtool command. The translated value of this variable is the name of the required link library.

15.7.3 The FHiCL File `readGens1.fcl`

There is only one fragment of `readGens1.fcl` that contains any new ideas. It is the fragment that configures the module label `read`, reproduced in Listing 15.4

Listing 15.4: Configuring the module label `read` in `readGens1.fcl`

```
1
2     read : {
3       module_type          : ReadGens1
4       genParticlesInputTag : "evtgen"
5     }
```

On line 4 of this fragment, the parameter `genParticlesInputTag` specifies the input tag that identifies the data product to be read by this exercise.

We recommend that you always initialize input tags using parameters from the parameter set and that you never initialize them using strings defined within the code. This will allow you run the same module on data products with different input tags; this is a widely used feature.

We further recommend that you not provide a default value in the call to get the parameter value from the parameter set. This derives from a general recommendation that parameters affecting physics output should never have default values; the only parameters with default values should be those that control debugging and diagnostics.

15.8 The Second Version, `ReadGens2`

Version 2 of this exercise consists of the files `ReadGen2_module.cc` and `readGen2.fcl`. To run this version, cd to your build directory and type the command:

```
art -c fcl/ReadGenParticles/readGens2.fcl >& output/readGens2.log
```

It will produce the same output as the previous two versions.

The only significant change from version 1 to version 2 is that lines

```
art::Handle<GenParticleCollection> gens;
event.getByLabel(gensTag_,gens);
```

have been replaced by the single (long) line:

```
art::ValidHandle<GenParticleCollection> gens =
    event.getValidHandle<GenParticleCollection>(gensTag_);
```

This version is a little verbose but that aspect will be addressed in version 3. Note that the class template `art::Handle` has been replaced by a new class template `art::ValidHandle`. Both class templates are defined in the same header file, `$ART_INC/art/Framework/Principal/Handle.h`.

The above line has functionality very similar to that of the two lines from version 1: the net result is that `gens` can be used as a pointer to the requested data product. It also has an interface to access the `art::Provenance` and the `art::ProductID`.

However, there are several signficant differences between `art::Handle<T>` and `art::ValidHandle<T>`:

1. Unlike an `art::Handle<T>`, which may be either valid or invalid, an `art::ValidHandle<T>` is guaranteed to be valid. It cannot be default-constructed.

2. A call to `getValidHandle<T>` will either return a properly constructed `art::ValidHandle<T>` or it will throw a ProductNotFound exception.

3. `art::ValidHandle` does not have an `isValid()` method.

4. Everytime that you use an `art::Handle<T>` as a pointer, it first checks that the pointer is valid. On the other hand, when you use an `art::ValidHandle<T>` as a pointer, no check is necessary; using an `art::ValidHandle<T>` as a pointer is as fast as using a bare pointer or a reference.

15.9 The Third Version, `ReadGens3`

Version 3 of this exercise consists of the files `ReadGen3_module.cc` and `readGen3.fcl`. To run this version, cd to your build directory and type the command:

```
art -c fcl/ReadGenParticles/readGens3.fcl >& output/readGens3.log
```

It will produce the same output as the previous two versions.

The only change from version 2 is that the call to `getValidHandle` has a slightly different syntax that provides the same behavior but is less verbose (shown here on two lines):

```
auto gens =
     event.getValidHandle<GenParticleCollection>(gensTag_);
```

This version uses a feature of C++ that is new in C++-11, the keyword `auto`. This keyword tells the C++ compiler to automatically determine the correct type for `gens`.

When you call the member function `getValidHandle<T>` the return type will always be `art::ValidHandle<T>`.

Version 3 is the version that we recommend you use but you can use any of the three. We introduced the version using the keyword `auto` as the last version because it is a handy shorthand when you know how to determine the correct type but it is very confusing if you do not know how to do so.

In future exercises we will use the pattern of version 3 regularly.

15.10 Suggested Exercises

Edit `readGens3.fcl` and supply the full input tag:
`genParticlesInputTag : "evtgen::exampleInput"`
Run *art* and observe that it works correctly.

Edit `readGens3.fcl` files and misspell the the requested module label, for example
`genParticlesInputTag : "genevent"`
Run *art* and observe the warning messages, which should look like the message in Listing 15.5. Note that the warning message includes information about the module label (read), the `module_type` (ReadGens3) and the fields of the requested data product. Observe that, for each event, *art* prints the warning message and continues with the next event.

Look at the last line of the *art* output and observe that *art* completed with status 0! This is because *art* treats ProductNotFound as a warning, not as an error that will initiate a shutdown.

Listing 15.5: Warning message for misspelled module label of data product

```
1
2  %MSG
3  %MSG-w FailModule:  ReadGens3:read 15-Jun-2014 10:00:24 CDT
4                                          run: 1 subRun: 0 event: 5
5  Module failed due to an exception
6  ---- ProductNotFound BEGIN
7    getByLabel: Found zero products matching all criteria
8    Looking for type: std::vector<tex::GenParticle>
9    Looking for module label: genevent
10   Looking for productInstanceName:
11
12 ---- ProductNotFound END
```

You can reconfigure *art* so that a ProductNotFound exception will cause *art* to shutdown gracefully. To do this, edit your modified `readGens3.fcl` and add the following line inside the `services` parameter set:
`scheduler : { defaultExceptions : false }`
This line tells *art* that its response to all exceptions should be to attempt a graceful shutdown. When you rerun *art* you should see output like that shown in Listing 15.6.

Listing 15.6: Exception message for `ProductNotFound`, default Exceptions disabled

```
%MSG-s ArtException:  PostCloseFile 15-Jun-2014 10:32:55 CDT PostEndRun
cet::exception caught in art
---- EventProcessorFailure BEGIN
  An exception occurred during current event processing
  ---- EventProcessorFailure BEGIN
    An exception occurred during current event processing
    ---- ScheduleExecutionFailure BEGIN
      ProcessingStopped.

      ---- ProductNotFound BEGIN
        getByLabel: Found zero products matching all criteria
        Looking for type: std::vector<tex::GenParticle>
        Looking for module label: genevent
        Looking for productInstanceName:

        cet::exception going through module ReadGens3/read
                                            run: 1 subRun: 0 event: 1
      ---- ProductNotFound END
      Exception going through path end_path
    ---- ScheduleExecutionFailure END
  ---- EventProcessorFailure END
  cet::exception caught in EventProcessor and rethrown
---- EventProcessorFailure END
%MSG
```

15.11 Review

In this chapter you have learned:

1. the type `tex::GenParticleCollection`
2. the four-part identifier of data product and the class `art::InputTag`
3. the class templates `art::Handle` and `art::ValidHandle`
4. how to get a handle to a data product, given its type and input tag
5. how to use a handle as a pointer to the requested data product
6. how to recognize a ProductNotFound warning message
7. how to tell *art* to treat the ProductNotFound exception as a hard error that will

initiate a graceful shutdown.

16 Exercise 7: Making a Histogram

16.1 Introduction

One of the workhorse tools of HEP data analysis is ROOT. Among its many features are tools for data analysis, visualization, presentation and persistency. As was discussed in Section 3.6.9, *art* uses ROOT as a tool for persistency of event-data *. In the code base of a typical HEP experiment there are many modules that use ROOT to create histograms, graphs, ntuples and trees, all of which are objects used for data analysis, visualization and presentation.

This exercise will show you how to use ROOT in the *art* environment using the ROOT class `TH1D` — one of many — as an example. Using this class you will create, fill and present 1-dimensional histograms. You can follow the model presented here if you wish to use related ROOT classes, such as the other histogram classes and the classes for graphs, ntuples and trees. If you are not familiar with graphs, ntuples and trees, examples will be given in future exercises.

Detailed information about ROOT is available from its website,
http://root.cern.ch/drupal.

Most of the modules that get run in a typical *art* job — plus *art* itself — use ROOT. Due to the way ROOT and *art* interact (a topic beyond the present scope), *art* needs to provide a mechanism to ensure that your module's use of ROOT will not interfere with the use of ROOT by *art* or by other modules running in the same job. The mechanism is an *art* service called `TFileService`, which does the necessary organizational work.

* You may have already guessed this, having seen the module types `RootInput` and `RootOuput` in previous exercises and the file type `.root` for the names of input and output event-data files.

This chapter will introduce you to *art* services in general and to the `TFileService` in particular.

All user interactions with ROOT should happen via this service.

Note that is possible to use ROOT as an event-processing framework, e.g., the AliRoot framework used by the ALICE Collaboration. But if you are using *art*, then *art* is always the event-processing framework and ROOT is used as a toolkit. The AliRoot documentation is at http://aliweb.cern.ch/Offline/AliRoot/Manual.html.

16.2 Prerequisites

Prerequisites for this chapter include all of the material in Part I (Introduction) and the material in Part II (Workbook) up to and including Chapter 15.

16.3 What You Will Learn

In this exercise you will learn:

1. What the `art::TFileService` is and what it does for you.
2. How to configure the `art::TFileService`.
3. What an `art::ServiceHandle` is and what it does for you.
4. How to access ROOT via the `art::TFileService`.
5. How to create and fill a ROOT `TH1D` histogram.
6. How to use the interactive ROOT browser to view the histogram.
7. How to run a CINT script to view the histogram and to write the histogram to a PDF file.
8. The naming convention used by the Workbook to distinguish event-data ROOT files from ROOT files containing histograms, ntuples, and so on. This convention is specific to the Workbook and it may differ from what your experiment uses.

16.4 Setting up to Run this Exercise

To run this exercise, you need to be logged in to the computer on which you ran Exercise 2 (in Chapter 10).

> If you are continuing on from a previous exercise, you need to keep both your source and build windows open.
>
> If you are logging back in, follow the instructions in Section 10.5 to reestablish your source and build windows.
>
> In your source window, cd to your source directory.
>
> ```
> cd $ART_WORKBOOK_WORKING_BASE/\
>   <username>/workbook/art-workbook
> ```
>
> Then cd to the directory for this exercise and look at its contents:
>
> ```
> cd art-workbook/FirstHistogram
> ```
>
> ```
> ls
> CMakeLists.txt     FirstHist1_module.cc
> drawHist1.C        firstHist1.fcl
> ```

The module `FirstHist1_module.cc` is very much like the module `ReadGens3_module.cc` from the previous exercise. The main difference is that it does not create any printout but rather, it fills a histogram displaying the number of generated particles in each event.

The FHiCL file `firstHist1.fcl` is very much like the file `readGens3.fcl` from the previous exercise. The important difference here is that `firstHist1.fcl` configures the `TFileService`.

The file `drawHist1.C`, discussed in Section 16.11, is a script written in a ROOT-defined language called CINT. This script contains the commands to open a ROOT file, draw a

histogram and write it to a PDF file.

The file `CMakeLists.txt` plays its usual role telling the build system what to do. Compared to the corresponding file for the previous exercise, it has two additional link libraries and contains an explicit directive that `drawHist1.C` should not be built. The meaning of this will become clear in the full discussion of `CMakeLists.txt`.

16.5 The Source File `FirstHist1_module.cc`

The C++ source code for this exercise is found in the file `FirstHist1_module.cc`. Open the file in your source window to see it as a whole. Listing 16.1 contains a fragment of this file, showing the included headers and the declaration of the module class `FirstHist1`. Compared to the file `ReadGens3_module.cc` from the previous exercise, four new lines have been added; they appear in the listing as:

1. line 7, which includes the header for the *art* `TFileService`
2. line 9, which includes the header for the ROOT class `TH1D`
3. line 22, which declares the member function `beginJob`
4. line 29, which declares a new member datum, named `hNGens_`, of type *pointer* to an object of type `TH1D`.

The name `hNGens_` was chosen because this pointer will eventually point at a histogram object that contains a histogram of the number of generated particles per event.

The *art* workbook has adopted the style that all names for pointers to histograms begin with the lower case letter “h”.

The two new headers can be found at:
`$ART_INC/art/Framework/Services/Optional/TFileService.h`
`$ROOT_INC/TH1D.h`

Listing 16.1: Declaration of the class `FirstHist1` from `FirstHist1_module.cc`

```
1
2  #include "toyExperiment/MCDataProducts/GenParticleCollection.h"
3
4  #include "art/Framework/Core/EDAnalyzer.h"
5  #include "art/Framework/Core/ModuleMacros.h"
6  #include "art/Framework/Principal/Event.h"
7  #include "art/Framework/Services/Optional/TFileService.h"
8
9  #include "TH1D.h"
10
11 #include <iostream>
12 #include <string>
13
14 namespace tex {
15
16   class FirstHist1 : public art::EDAnalyzer {
17
18   public:
19
20     explicit FirstHist1(fhicl::ParameterSet const& );
21
22     void beginJob() override;
23     void analyze(art::Event const& event) override;
24
25   private:
26
27     art::InputTag gensTag_;
28
29     TH1D* hNGens_;
30
31   };
32
33 }
```

The conventions for including header files from ROOT differ from those for including header files from *art* and from toyExperiment. To remind you, the conventions for *art* and the toyExperiment UPS product are:

1. The names of all classes and functions are inside a namespace, `art` or `tex`, respectively.
2. In the header file `#include` lines, the name of the package to which the header belongs is always the first element of the path.

When ROOT was developed, namespaces were not supported robustly by many C++ compilers. Therefore a different set of conventions were adopted – and remain – for ROOT:

1. The names of all ROOT classes and functions are in the *global* namespace, i.e., they are not part of a namespace defined by ROOT.
2. The names of all ROOT classes begin with a capital letter T followed by an upper case letter (this serves as a weak substitute for using a namespace).
3. The syntax to include a file from ROOT is to give the filename without any leading path elements. The clue that the file is a ROOT header file comes from the leading capital T.

Listing 16.2 shows the implementation section of the file `FirstHist1_module.cc`.

The new features in this listing are:

1. line 5, which initializes `hNGens_` to have the value of a null pointer
2. lines 8 through 14, which create an empty histogram
3. line 20, which fills the histogram with the number of generated particles in the current event

The identifier `nullptr`, used in line 5, was added to the C++ core language in the 2011 Standard. It is the value of a pointer that points to nothing; in practice it has a value of zero. You will very likely encounter code written prior to the 2011 Standard. In such code you will see the equivalent of line 5 written in one of the following two ways: `hNGens_(0)` or `hNGens_(NULL)`. In the second form, the value `NULL` is a C-Preprocessor MACRO variable that is defined to have a value of 0.

Listing 16.2: Implementation of the class `FirstHist1`

```
1
2  tex::FirstHist1::FirstHist1(fhicl::ParameterSet const& pset ):
3    art::EDAnalyzer(pset),
4    gensTag_(pset.get<std::string>("genParticlesInputTag")),
5    hNGens_(nullptr){
6  }
7
8  void tex::FirstHist1::beginJob(){
9
10   art::ServiceHandle<art::TFileService> tfs;
11   hNGens_ = tfs->make<TH1D>( "hNGens",
12             "Number of generated particles per event", 20, 0., 20.);
13
14 }
15
16 void tex::FirstHist1::analyze(art::Event const& event ){
17
18   auto gens = event.getValidHandle<GenParticleCollection>(gensTag_);
19
20   hNGens_->Fill(gens->size());
21
22 }
```

We strongly recommend, first, that you use `nullptr` for this purpose, and second that you never use the C-Preprocessor `NULL`.

16.5.1 Introducing `art::ServiceHandle`

Section 3.6.5 discussed the idea of *art* services. These are classes that provide some functionality (i.e., a service) that can be used by any module or by other *art* services. In this exercise you will see your first example of an *art* service, the `art::TFileService`, which provides a bookkeeping layer to ensure that your use of ROOT does not interfere with other uses of ROOT within the same *art* job.

In a similar way that access to data products is provided by the class templates `art::Handle` and `art::ValidHandle`, access to services is provided by the class template `art::ServiceHandle`. Line 10 in Listing 16.2 tells the compiler to default construct an object, named `tfs`, of type `art::ServiceHandle<art::TFileService>`. The constructor of `tfs` will contact the internals of *art* and ask *art* to find a service of type `art::TFileService`. If *art* can find such a service, it will give the service handle a pointer to the service. If not, it will throw an exception and attempt a graceful shutdown.

Once a service handle has been constructed, the downstream code can use the service handle as a pointer to the pointee, i.e., to `art::TFileService`.

The header file for `art::ServiceHandle` is found at:
`$ART_INC/art/Framework/Services/Registry/ServiceHandle.h`
It is automatically included by one of the files that are already included in `FirstHist1_module.cc`.

16.5.2 Creating a Histogram

Lines 10 through 12 of Listing 16.2, use `art::TFileService` to create a new histogram object of type `TH1D`. In the call to the member function template `tfs->make` (lines 11 and 12), the type of object to be created is specified using a template argument. The function arguments, listed below, are the arguments needed by a constructor of that type of object. You do not need to understand why things are done this way or how it all works. You just need to follow the pattern. The return value of the call to `tfs->make`

Listing 16.3: Creating histogram object of type `TH1D`

```
10  art::ServiceHandle<art::TFileService> tfs;
11  hNGens_ = tfs->make<TH1D>( "hNGens",
12            "Number of generated particles per event", 20, 0., 20.);
```

is a pointer to the newly created histogram object and this value is assigned the member datum `hNGens_`.

In the case of creating a `TH1D`, the five function arguments are:

1. the name by which ROOT will know this histogram; the *art* workbook has adopted the convention that this name will always be the name of the corresponding member datum, excluding the underscore (in this case `hNGens`)
2. the title that will be displayed when the histogram is drawn (given on line 12 of Listing 16.2)
3. the number of bins in the histogram (20)
4. the lower edge of the lowest bin of the histogram (0.)
5. the upper edge of the uppermost bin of the histogram (20.)

ROOT defines that the low edge of a bin is within that bin, while the upper edge of a bin is part of the next bin up. Therefore the lower edge of the lowest bin is inside the histogram but the upper edge of the uppermost bin is outside of the histogram.

If you would like to learn more about the `TH1D` class you can look at its header file or you can read about it on the ROOT web site: http://root.cern.ch/root/html534/TH1D.html.

Where is the histogram created? The histogram is created in memory that is owned and managed by ROOT. ROOT also knows that when the job is finished, it should write the histogram to a ROOT output file that you can inspect at a later time. The name of the output file is specified in the FHiCL file for the *art* job; more on that later. We will call this file the *histogram output file* or *histogram file*. Although histogram files often contain much more than just histograms, the name is in fairly common usage among the experiments that use *art*. Histogram files do not contain *art* data products.

Just as file systems have the notion of directories and subdirectories (or folders and subfolders if you prefer), a ROOT file has the notion of directories and subdirectories

Listing 16.4: Filling the histogram

```
20 hNGens_->Fill(gens->size());
```

that are internal to the ROOT file. If a module makes at least one histogram, then the `TFileService` will first create a new top-level directory in the histogram file. The name of this top-level directory is the name of the module label of the module that created the histogram. All ROOT objects that are created by that module will be created within this top level directory. When the contents of ROOT-managed memory are written to the histogram file, this directory structure is preserved.

Recall that within a given *art* job each module label must be unique. This ensures that, for every module instance that uses the `TFileService`, a uniquely named top-level directory will be created in the histogram file. It is this strategy that ensures that the histogram names of my module will never collide with the histogram names of your module.

16.5.3 Filling a Histogram

Line 20 of Listing 16.2, fills the histogram pointed to by `hNGens_` with the number of generated particles for this event.

If you look up the function prototype for `TH1D::Fill` you will see that it expects an argument that is a double. On the other hand, `gens->size()` returns an unsigned integer. One of the features of C++ is that it can automatically convert the unsigned integer to a double and pass that to the function. For details consult the standard C++ documentation that is listed in Section 6.7.

16.5.4 A Few Last Comments

All of the comments above about management of ROOT directories and writing histograms to files are also true for most other sorts of ROOT objects. In particular they are true for `TTrees` and `TNtuples`.

If you think carefully about `FirstHist1_module.cc` you might wonder why there is no `endJob` member function containing a call to delete the histogram that was created in

the `beginJob` member function. The answer is that when you create a histogram that is controlled by ROOT, then ROOT is responsible for calling delete at the right time.

If you talk to an HEP old-timer about creating histograms, he or she will probably call it "booking a histogram." This is language left over from a precursor to ROOT named HBOOK.

By convention both the histogram files (output) and the *art* event-data files (input) end in `.root`. Even though both are ROOT files, the two types of files are structured very differently and are not in any way interchangeable or interoperable.

The *art* Workbook has adopted the convention that *art* event-data files always end in `_data.root`. All other files ending in `.root` are histogram files.

Some experiments have adopted a similar convention while others have adopted precisely the opposite convention: files ending in `_hist.root` are histogram files and all other files ending in `.root` are *art* event-data files.

16.6 The Configuration File C++ firstHist1.fcl

The file `firstHist1.fcl`, shown in Listing 16.6, is very much like the file `readGens3.fcl` from the previous exercise.

The most important new feature is at line 13,

Listing 16.5: `TFileService` in `firstHist1.fcl`

```
13 TFileService : { fileName : "output/firstHist1.root" }
```

which configures the `TFileService`. This service has one required parameter, which is the name of the histogram file that contains the histograms, trees, and so on that are created by the *art* job.

If this parameter is missing, or if the configuration for the `TFileService` is missing entirely, then the first attempt to get a service handle to the `TFileService` will throw an exception, and *art* will attempt a graceful shutdown.

Unlike in the previous excercises, the FHiCL file runs on the large input event-data file, `inputFiles/input04_data.root`, which contains 1000 events.

Listing 16.6: `firstHist1.fcl`

```
1  #include "fcl/minimalMessageService.fcl"
2
3  process_name : firstHist1
4
5  source : {
6    module_type : RootInput
7    fileNames   : [ "inputFiles/input04_data.root" ]
8  }
9
10 services : {
11   message : @local::default_message
12   TFileService  : { fileName : "output/firstHist1.root" }
13 }
14
15 physics :{
16   analyzers: {
17     hist1 : {
18       module_type          : FirstHist1
19       genParticlesInputTag : "evtgen"
20     }
21   }
22
23   e1        : [ hist1 ]
24   end_paths : [ e1 ]
25
26 }
```

16.7 The file `CMakeLists.txt`

The `CMakeLists.txt` file used for this exercise is shown in Listing 16.7. Compared to the corresponding file for the previous exercise, there are two new features.

1. Two link libraries have been added.
2. There is a directive indicating that `cmake` should not build files ending in `.C`.

The two new link libraries are specified by `${ART_FRAMEWORK_SERVICES_OPTIONAL_TFILESERVICE_SERVICE}` and `${ROOT_HIST}` (lines 12 and 15). These items are both `cmake` variables that were defined for you when established the development environment. The first variable translates to

Listing 16.7: `CMakeLists.tex` in the directory `FirstHistogram`

```
1  file( GLOB ROOT_MACROS_DO_NOT_BUILD
2        RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} *.C )
3  art_make(
4    EXCLUDE ${ROOT_MACROS_DO_NOT_BUILD}
5    MODULE_LIBRARIES
6      ${TOYEXPERIMENT_MCDATAPRODUCTS}
7      ${ART_FRAMEWORK_CORE}
8      ${ART_FRAMEWORK_PRINCIPAL}
9      ${ART_PERSISTENCY_COMMON}
10     ${ART_FRAMEWORK_SERVICES_REGISTRY}
11     ${ART_FRAMEWORK_SERVICES_OPTIONAL}
12     ${ART_FRAMEWORK_SERVICES_OPTIONAL_TFILESERVICE_SERVICE}
13     ${FHICLCPP}
14     ${CETLIB}
15     ${ROOT_HIST}
16 )
```

`$ART_LIB/libart_Framework_Services_Optional_TFileService_service.so` and the second translates to `$ROOTSYS/lib/libHist.so`, which contains the object code for the class `TH1D`, among others.

Many projects use the convention that files ending in `.C` contain code written in the **C** programming langauge. By default `cmake` will assume that files ending in `.C` follow this convention and, therefore, it will try to compile and link them. We have already encountered a CINT script that ends in `.C`. The `cmake` needs to ignore CINT files. `CMakeLists.txt` includes code to effect this.

You do not need to understand the details of how the `CMakeLists.txt` excludes `drawHist1.C` from the build. For those who wish too look up the details, the high level explanation follows. Lines 1 and 2 in the listing of `CMakeLists.txt` tell `cmake` to define a new `cmake` variable named `ROOT_MACROS_DO_NOT_BUILD`. This variable is the set of all filenames that end in `.C` from the same directory as the `CMakeLists.txt` file. Line 4 in `CMakeLists.txt` tells `cmake` that it should do nothing for all files that appear in the translation of this variable.

16.8 Running the Exercise

To run this exercise, cd to your build directory and run *art*:

```
cd $ART_WORKBOOK_WORKING_BASE/<username>/\
  workbook/build-prof

art -c fcl/FirstHistogram/firstHist1.fcl
```

This module does not make any of its own printout. You should see the standard printout from *art*, including the final line saying that *art* will exit with status 0. Remember to add >& output/<filename>.log to the end of the command to send the printout to a file. The Workbook will not always show this in subsequent exercises, but it is always recommended.

You should see that the *art* job created the file `output/firstHist1.root`. This is the histogram file.

16.9 Inspecting the Histogram File

In this section you will inspect the file `output/firstHist1.root`.

First, look again at `fcl/FirstHistogram/firstHist1.fcl`. Note that the module label of the `FirstHist1` module is `hist1`.

To inspect the histogram you will remain in your build directory and you will run the interactive ROOT program, using the command root. This command was put into your path when you established your build environment. To perform this exercise:

1. Enter:

   ```
   root -l output/firstHist1.root
   ```

   The command line option is a lower case letter L. In your build window, some output and a new prompt will appear:

```
root [0]
Attaching file output/firstHist1.root as
_file0...
root [1]
```

2. At the root prompt, type the command

   ```
   TBrowser* b = new TBrowser("Browser", _file0);
   ```

   This will open a new window on your display; a screen capture of this window is shown in Figure 16.1. We will refer to this window as the TBrowser window.

3. In the left hand panel of the TBrowser window, you will see a ROOT file icon followed by the name of the file `output/firstHist1.root`. Double click on this line.

4. This will create a new line in the left hand panel of the TBrowser window. The line contains a folder icon followed by the name of a folder, `hist1;1`. Double click on this line.

5. This will create another new line in the left hand panel of the TBrowser window. This line contains a blue histogram icon and the name of a histogram `hNGens;1`. Double click on this line.

6. The histogram will appear in the right hand panel of the TBrowser window. Figure 16.2 shows a screen capture of the window with the histogram drawn.

7. To exit root, return to the build window and, at the root prompt, type .q (a period followed by a lower case letter Q).

   ```
   .q
   ```

8. Another way to quit root is to click on the "Browser" pull-down menu on the top bar of the TBrowser window. From the menu select "Quit ROOT".

In step 4 you should have recognized the name of the folder, `hist1;1`. Ignoring the trailing `;1`, it is the name of the module label used in `firstHist1.fcl`. In step 5 you should have recognized the name of the histogram, `hNGens`; ignoring `;1`, it is the name that you gave the histogram when you created it.

About the `;1` that ROOT has stitched onto `hist1` and `hNGens`: ROOT calls these *cycle numbers*. They are part of a checkpointing mechanism that is beyond the scope of this exercise; if you ever see more than one cycle number for a ROOT object, the highest number is the one that you want. Consult the ROOT documentation for more details.

Now look at the histogram in the right hand panel of the TBrowser window. In the statistics box on the upper right you should see that it has 1000 entries, one for each event in the input file. You should also notice that only the odd bins are populated: this is because the generated events always contains three signal particles, plus a random number of pairs of background particles ($3 + 2n$). The three signal particles are φ meson and the two kaons into which it decays. You should also recognize the title and the name that you set when you created the histogram. Finally you should recognize that the binning matches the binning you requested when you created the histogram.

16.10 A Short Cut: the browse command

The above description for viewing a ROOT file interactively requires a lot of tedious typing at step 2. The toyExperiment UPS product provides a command named browse that does the typing for you. To use this command:

```
browse output/firstHist1.root
```

Then follow the instructions from the previous section, starting at step 3.

When you created your *art* build environment, the toyExperiment UPS product put the command browse into your path. This command is implemented as a bash script and you

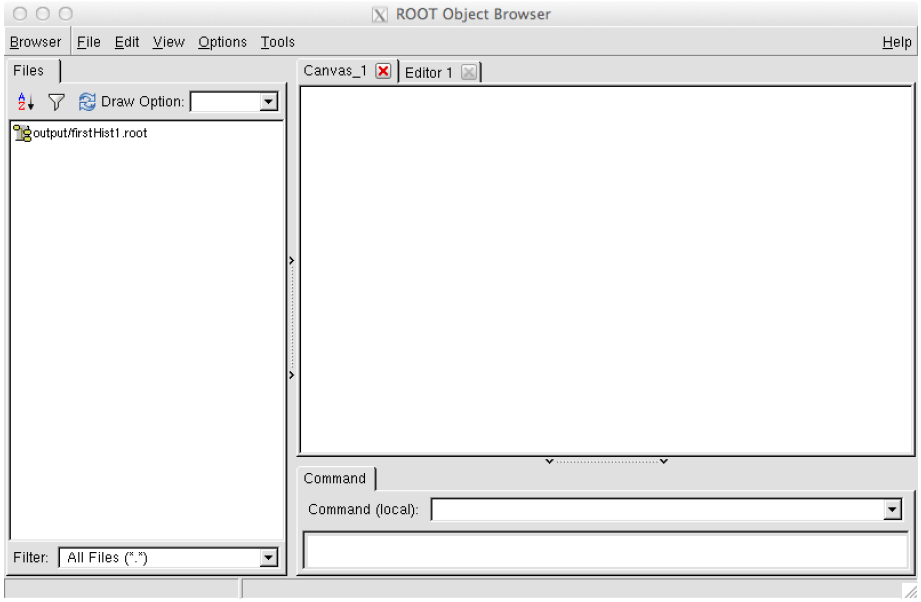


Figure 16.1: The TBrowser window immediately after opening `output/firstHist1.root`.

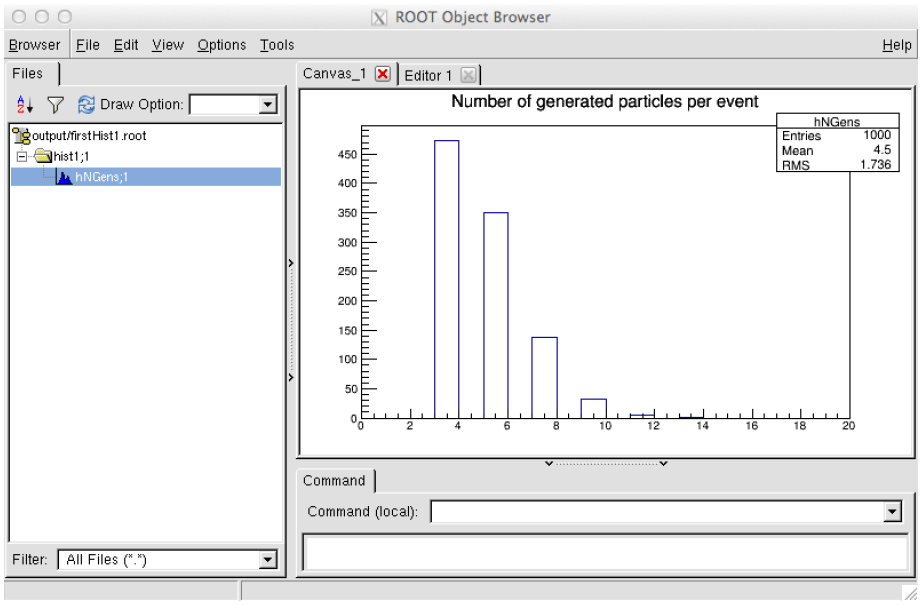


Figure 16.2: TBrowser window after displaying the histogram `hNGens;1`.

can find its definition using the following bash command:

```
type browse
browse is <...>/scripts/browse
```

where `<...>` will change from one site to the next; it will have the value of `$TOYEXPERIMENT_DIR` as defined at that site.

```
[aheavey@cluck build-prof]$ type browse
browse is hashed (/home/kutschke/products//toyExperiment/v0_00_15/scripts/b
[aheavey@cluck build-prof]$ echo $TOYEXPERIMENT_DIR
/home/kutschke/products//toyExperiment/v0_00_15
[aheavey@cluck build-prof]$
```

16.11 Using CINT Scripts

When you type a command at the root prompt, you are typing commands in a ROOT-defined language called CINT. Scripts in this language are sometimes called ROOT scripts and other times CINT scripts.[†] As with many interpreted languages, you may write CINT commands in a file and execute that file as script.

It is a common convention that files that contain CINT scripts have a file type of `.C`. This convention is followed throughout the *art* workbook.

This exercise provides an example of a CINT script, `drawHist1.C`, in Listing 16.8.

To use this script, run:

```
root -l drawHist1.C
```

This will open a window on your display, draw the histogram in that window and save the window to the PDF file `output/NumberGenerated.pdf`, which is shown as Figure 16.3. When the script is complete, it returns control to the root prompt in your build window. At this prompt you can issue more CINT commands. To exit ROOT, type `.q` at the root prompt.

[†] *CINT* is a (somewhat misleading) acronym for C++ INTerpreter. While CINT will correctly execute a lot of C++ code, there is legal C++ code that is not legal CINT and there is legal CINT that is not legal C++. There is also code that is legal in both C++ and CINT but that does subtly different things in the two environments. Therefore it is more correct to say that the CINT language shares a lot of syntax with C++, not that it *is* C++.

Listing 16.8: Sample CINT file `DrawHist1.C`

```
//
// Root script to draw the histogram made by FirstHist1_module.cc
//

{

  // With this you can reinvoke the script without exiting root
  // and restarting.
  gROOT->Reset();

  // Get rid of grey background (ugly for printing).
  gROOT->SetStyle("Plain");

  // Recommended content of statistics box:
  // Number of Entries, Mean, Rms, Underflows, Overflows,
  //  Integral within limits
  gStyle->SetOptStat("emruoi");

  // Open the input file that contains histogram.
  TFile* file = new TFile( "output/firstHist1.root");

  // Get pointer to the histogram.
  TH1D* hNGens;  file->GetObject("hist1/hNGens",  hNGens);

  // Open a new canvas on the screen.
  TCanvas *canvas = new TCanvas("canvas", "Plots from Firsthist1.root" );

  // "H9": draw outline histogram ("H") in high resolution mode (9)
  hNGens->Draw("H9");

  canvas->Update();
  canvas->Print("output/NumberGenerated.pdf");

}
```

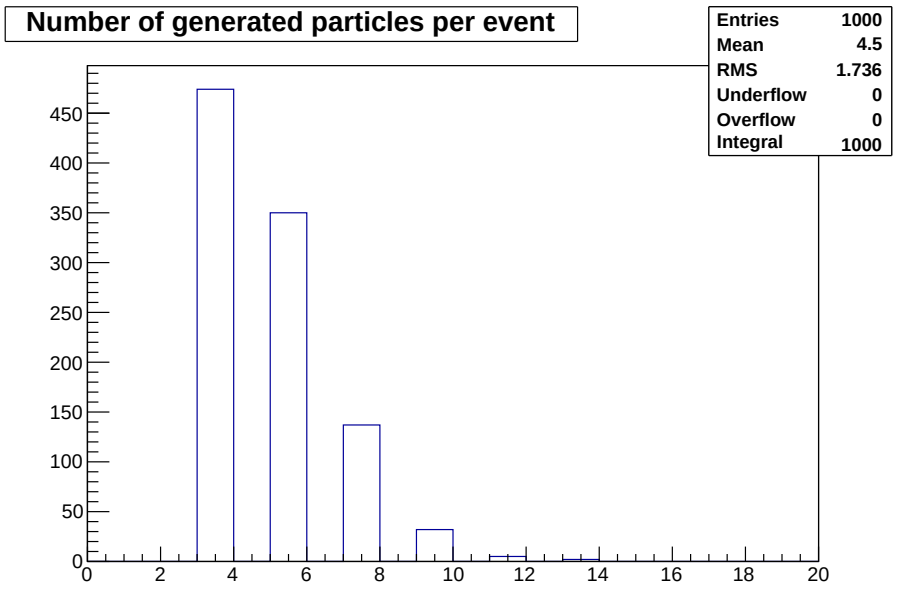


Figure 16.3: Figure made by running the CINT script `drawHist1.C`.

If you compare this figure to the histogram in Figure 16.2 you will see that there is difference in the statistics box in the upper right. In this figure, the name of the histogram is not shown but three new fields are: the number of entries below the lower limit (Underflow), the number of entries above the upper limit (Overflow) and the number of entries between the limits (Integral). The field named "Entries" is the sum of Integral plus Underflow plus Overflow.

It is beyond the scope of this writeup to describe all of the features in Listing 16.8, but we will describe some of the code.

Regarding lines 9 and 12, we will let comments in the code be a guide, and give two additional hints. The object `gROOT` is a pointer to an instance of the class `TROOT` and the object `gStyle` is a pointer to an instance of the class `TStyle`. To find the documentation for these classes, see Section 16.12.

Line 17 tells ROOT what to draw in the statistics box in the upper right of every histogram. The comments in the code describe the mnemonics of the letter codes. The full set of letter codes is described on the ROOT web site: http://root.cern.ch/root/html534/TStyle.html as part of the documentation for the member function `SetOptStat`.

Line 20 opens the input file. The ROOT type `TFile` is ROOT's interface to ROOT information that lives in a disk file; it allows a ROOT program to write ROOT objects to the file and read ROOT objects from the file. You can learn more about the class `TFile` from the ROOT web site http://root.cern.ch/root/html534/TFile.html.

Line 23 has two statements on it. The first declares `hNGens` as an object of type pointer to `TH1D` and initialzies it to 0 (ROOT does not support nullptr). The second asks the `TFile` to find a ROOT object named "hist1/hNGens", copy it from the file into memory and to set the pointer `hNGens` to point to that object. If ROOT cannot find the requested object, or if the type of the requested object does not match the type of the pointer, ROOT will leave `hNGens` with a value of 0. This is reminiscent of asking an `art::Event` to fill an `art::Handle`.

Line 26 tells ROOT to open a new window on your display. The first argument is an arbitrary name that must be unique within the job; ROOT uses it internally to differentiate multiple canvases. The second argument is the title that will be drawn on the title bar of the window.

Line 29 tells ROOT to draw the histogram on the canvas. If, at line 21, ROOT was unable to properly set the pointer, then this line will produce an error message and return control to the root prompt in the build window.

Line 31 tells ROOT to flush its internal buffers and make sure that everything that is in the queue to be drawn on the canvas is actually drawn.

Line 33 tells ROOT to save the canvas by writing it to the file specified as the function argument. The format in which the file will be written is governed by the file type field in the filename, `.pdf` in this case. Many other formats are supported and a full list is available at: http://root.cern.ch/root/html534/TPad.html#TPad:Print

16.12 Finding ROOT Documentation

The main ROOT web site is http://root.cern.ch/drupal. On the top navigation bar there is a title labeled *Documentation*. Hover over this and a pull-down menu will appear. From this menu you can find links to Tutorials, How To's FAQs, a User Guide, a Reference Manual and more. Some of the documentation is version-dependent. The version of ROOT used by this version of the workbook is v5.34/18.

One possible starting point for learning ROOT is the ROOT Primer. You can find it by first going to the User's Guide page or you can follow the direct link:
http://root.cern.ch/drupal/content/users-guide#primer

One of the most useful parts of the ROOT documentation suite is the Reference Guide, which can be reached from the pull down menu. The direct link to this page is:
http://root.cern.ch/root/html534/ClassIndex.html
This section has a description of all of the members in each ROOT class.

16.13 Suggested Activities

16.13.1 Overwriting Histogram Files

Suppose that you run *art* and make a histogram file. If you run *art* again, what happens? The answer is that it will overwrite the existing histogram file and replace it with the one created in the second job.

To illustrate this, rerun the exercise but tell *art* to only do 500 events:

```
art -c fcl/FirstHistogram/firstHist1.fcl -n 500
```

Inspect the histogram file and you will see that the histogram now has only 500 entries.

It is your responsibility not to overwrite files that you wish to keep. One way to keep a file that is valuable is to use the unix chmod command to change the protections on the file so that it is readonly:

```
chmod -w <filename>
```

To restore the file to a writeable state the unix command is:

```
chmod o+w <filename>
```

16.13.2 Changing the Name of the Histogram File

You can change the name of the histogram file by editing the FHiCL file but you can also do so from the *art* command line by using the `--TFileName` option; the short form of this option is `-T`.

> Run the two following commands, both shown here on two lines (note that in both cases a space is required before the backslash since a space exists in each command line at these points):
>
> ```
> art -c fcl/FirstHistogram/firstHist1.fcl --TFileName \
>   output/anotherName.root -n 400
> ```
>
> ```
> art -c fcl/FirstHistogram/firstHist1.fcl -T output/yetAnotherName.root \
>   -n 750
> ```
>
> After each run, inspect the ouptut file and verify that the number of entries in the histogram matches the number of events requested on the command line.

16.13.3 Changing the Module Label

In `firstHist1.fcl`, change the name of the module label, `hist1`. Rerun the job and browse the histogram file. You should see that the name of the directory in the histogram file has changed to match the new module label.

16.13.4 Printing From the TBrowser

You can use the `browse` command to open the histogram file and view the histogram.

> In the TBrowser window, click on the "File" button. This will open a pull-down menu.

Click on the line "Save As ...". This will open a dialog window that will let you save the histogram displayed on the canvas in a variety of formats, including `.png`, `.gif`, `.jpg` and `.pdf`.

16.14 Review

In this exercise you have learned:

1. How to configure the `TFileService`.
2. How to use an `art::ServiceHandle` to access the `TFileService`.
3. How to use the `TFileService` to create a histogram that will automatically be written to the histogram file.
4. Three different ways to view the contents of the histogram file: by launching a TBrowser by hand, by using browse command and by running a CINT script.
5. The convention used by the *art* Workbook to differentiate histogram files from *art* event data files.

17 Troubleshooting

17.1 Updating Workbook Code

If the remote machine that you log onto to run the Workbook exercises runs into problems during the setup procedure, it's possible that the admin for that machine has not installed the most recent versions of the Workbook code, or some dependent code. Contact the administrator.

17.2 XWindows (xterm and Other XWindows Products)

17.2.1 Mac OSX 10.9

The XWindows products, xterm, xclock and so on, likely reside in the directory `/opt/x11/bin/`. You will need to add this to your PATH. When your machine is connected to a second monitor, the XWindows products may not position properly on the screen. You may need to contact a Macintosh support person to configure the X11 setup properly so that it works with the multiple screen configuration.

At Fermilab, open a service desk ticket at `https://fermi.service-now.com/navpage.do`Service-Now.

Part III

User's Guide

18 Obtaining Credentials to Access Fermilab Computing Resources

To request your Fermilab computing account(s) and permissions to log into the your experiment's nodes, fill out the form Request for Fermilab Visitor ID and Computer Accounts. Typically, experimenters that are not Fermilab employees are considered *visitors*. You will be required to read the Fermilab Policy on Computing.

After you submit the form, an email from the Fermilab Service Desk should arrive within a week (usually more quickly), saying that your Visitor ID (an identifying number), Kerberos Principal and Services Account have been created. You will need to change the password for both Kerberos and Services.

18.1 Kerberos Authentication

Your Kerberos Principal is effectively a username for accessing nodes that run Kerberos in what's called the FNAL.GOV *realm* (all non-PC Fermilab machines). *

To change your Kerberos password, first choose one (minimum 10 characters with mixture of upper/lower case letters and numbers and/or symbols such as !, , #, $, ; &, *, %). From your local machine, log into the machine using `ssh` or `slogin` and run the `kpasswd` command. Respond to the prompts, as follows:

```
$ kpasswd <username>@FNAL.GOV

    Password for username@FNAL.GOV: <--- type your current password he
```

*The FERMI.WIN.FNAL.GOV realm is available for PCs.

```
New password:                    <--- type your new password here

New password (again):             <--- type your new password here for c

Kerberos password changed.
```

Your Kerberos password will remain valid for 400 days.

18.2 Fermilab Services Account

The Services Account enables you to access a number of important applications at Fermilab with a single username/password (distinct from your Kerberos username/password). Applications available via the Services Account include SharePoint, Redmine, Service Desk, VPN and others.

To get your initial Services Account password, a user must first contact the Service Desk to get issued a first time default password. Once a default password is issued, users can access http://password-reset.fnal.gov/ to change it.

If you are not on-site or connected to the Fermi VPN, call the Service Desk at 630-840-2345. You will be given a one-time password and a link to change it.

19 git

The source code for the exercises in the *art* workbook is stored in a source code management system called *git* and maintained in a repository managed by Fermilab. Think of *git* as an enhanced *svn* or (a VERY enhanced) *cvs* system. The repository is located at . You will be shown how to access it with *git*.

If you want some background on *git*, we suggest the Git Reference.

You will need to know how to install *git*, download the workbook exercise files initially to your system and how to download updates. You will not be checking in any code.

To install *git* on a Mac:

```
$ http://git-scm.com/download/mac
```

This will automatically download a disk image. Open the disk image and click on the .pkg file.

In your home directory, edit the file .bash_profile and add the line:

```
$ export PATH=/usr/local/git/bin/:${PATH}
```

```
$ git clone ssh://p-art-workbook@cdcvs.fnal.gov/cvs/projects/art-workb
```

and how to download updates as the developers make them:

```
$ git pull
```

19.1 Aside: More Details about git

To bring your working copy of the workbook code up to date, you need to use `git`. Before describing the required `git` commands, we need to explain a little more about how git works and how the art-workbook team have chosen to use it. If you are familiar with `git` you can skip this section.

19.1.1 Central Repository, Local Repository and Working Directory

At any given time, there are three copies of the code that you need to be aware of, the central repository, your local clone of the central repository and the working copy of the code in your source directory.

1. The central git repository that contains all of the versions of the workbook is hosted by a machine named `cdcvs.fnal.gov`* The art-workbook team updates this repository as it develops and maintains the exercises.

2. In section 10.4.1, in step 5b) you used the `git clone` command to make a copy of the central repository in your source directory. This clone contains a complete history of the development of art-workbook *as it existed at the time that you made the clone*. The local clone is found in the `.git` subdirectory of your source directory.

3. In section 10.4.1, in step 5d), you used the `git checkout` command to choose one of the tagged versions of art-workbook. This command looked into your local clone of the central repository, found all of the files in the requested version and put copies of them in the correct spot in the directory tree rooted at your source directory.

There are two other source code managment systems that are widely used in HEP, `cvs` and `svn`. If you are familiar with either of these, git has an extra level: the concept of a local clone of the central respostory does not exist in those systems. That is, when are using `cvs` or `svn` and you want to switch to another version of the code, you need to contact the central repository but, when you are using `git`, you need only to contact your local clone of the central repository.

* Originally this machine hosted only `cvs` repositories, hence its name. It now hosts `cvs`, `svn` and `git` repositories.

To bring your working code up to date you need to do two steps:

1. Update your local clone of the central repository.
2. Checkout the new version from the local clone.

The discussion of the checkout has several cases. Each is discussed in one of the following sub-sections. It is possible for all four of these cases to occur on any given checkout.

19.1.1.1 Files that you have Added

When you worked on Exericse 2, you added some files to your working directories; for example you added the files `Second_module.cc` and `second.fcl`. When you do the checkout of the new version, these files will remain in your working directory and will not be modifed; however the checkout command will generate some informational messages telling you that your working directories contain files that are not part of the checked out version.

You do not need to take any action; just be aware of the situation.

19.1.1.2 Files that you have Modified

Another case occurs for files that have the following properties:

1. They were part of the old version.
2. You have modified them.
3. They have not been modified in the central repository since you cloned the repository.

For example, suppose that you modified `first.fcl`; it is very unlikely that this file would have been modified in the central repository after you cloned the repository.

In this case, the checkout command will issue a warning message to let you know that your working version contains changes that are not part of the release you checked out.

You do not need to take any action; just be aware of the situation.

19.1.1.3 Files with Resolvable Conflicts

Another case occurs for files that have the first two properties from the list in Section 19.1.1.2 but which have been modified in the central repository since you cloned the repository. This will happen from time to time when when we update exercises based on suggestions from users.

When this happens there are two cases, one of which is discussed here, while the other is discussed in the next sub-section.

If the two sets of changes (yours and those in the repository) are on different lines of the file `git`, will usually successfully merge these changes; `git` will then issue an warning message telling you want it has done.

It is your repsonsibility to identify these cases, understand the changes made in the repository and understand if `git` did the merge correctly.

19.1.1.4 Files with Unresolvable Conflicts

The final case is a variant of the previous case; it occurs when `git` is unable to automatically merge conflicting changes. This will happen when the changes you made and the changes made in the repository affect the same line, or lines, of code. When `git` does not know how to merge the changes it will give up, add markup to the offending files to mark the conflict and issue an error message. This leaves the offending files in an unusable state and you must correct the conflicts, by hand, before continuing.

The art-workbook has been designed so that this should happen very, very rarely. Most readers should bookmark this spot for future reference and only read it when they need to.

19.1.2 `git` Branches

`git` supports a concept known as *branches*. This is a very powerful feature that simplifies the task of having many developers collaboratively working on a single code base. Moreover, different experiments can choose to use branches in different ways; therefore a full description of branches is very open ended topic.

Fortunately, to use the workbook you do not need to know very much about branches; all that you need to know is summarized in Figure 19.1, which shows a simplified view of the way that the art-workbook team uses `git` branches. In Figure 19.1 time starts at the bottom

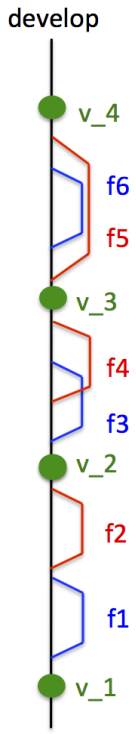


Figure 19.1: A figure to illustrate the idea of git branches, as used in the Workbook; the figure is described in the text.

of the figure and runs upward. The art-workbook team has adopted the convention that the most up to date version of the art-workbook code will always be found by checking out a branch named *develop*. In Figure 19.1 the vertical line represents the develop branch.

At the earliest time represented in Figure 19.1, the develop branch existed in some state that the art-workbook team liked. So they tagged the develop branch with the name `v_1`, for version 1. Shortly afterwards, the art team needed to add some improvements. To do this they did:

1. Use the `git pull` command to make sure that their local copy of the respository is up to date.

2. Check out the develop branch.
3. Start a new branch; in this example the new branch has the name `f1`, for "feature number 1".
4. Do all the development work on this branch. When they change files and commit their changes, the changes stay local to the branch.
5. Once the new code has been tested it is merged back into the develop branch.
6. Use the `git pull` command to make sure that their local copy of the respository is up to date; in this example, no one else has made.
7. Finally the developer must push their local copy of the respository to the central repository.

This is illustrated in Figure 19.1 by the blue line labelled `f1`. While the developer is working on `f1`, he can change back and forth between the develop branch and the `f1` branch.

In Figure 19.1 the red line labelled `f2` represents a second feature that is added to the code base following the same pattern as the first.

In this example, the development team decided to tag the develop branch after the `f2` branch was merged back in; the tag was given the name `v2`; this is represented by the green filled circle in the figure.

The next items on the figure are the branches named `f3` and `f4`. In this example, someone started with the develop branch and began work on the feature `f3`. A little later someone else (or maybe the same person) started with the develop branch and began work on the feature `f4`. The person starting work on `f4` did so before the changes from `f3` were merged back into the develop branch; therefore the two branches `f3` and `f4` both start from the same place, the `v2` tag of develop. In this example, the next item on the timeline is that the developer of `f3` commits their changes back to the develop branch. Sometime after that the developer of `f4` merges their changes back. At this time the developer of `f4` has the responsibility to check for conflicts that occurred during the merge and fix them; this may or may not require consultation with the author of `f3`.

After this, the develop branch is again tagged, this time with a version named `v_3`.

The next items on the timeline are the branches named `f5` and `f6`. This example was

included to show that it is legal for `f6` both start and end during the time that `f5` is alive.

Finally, the develop branch is tagged one more time, this time with the name `v_4`.

In Figure 19.1 consider a time when both branches `f5` and `f6` are active.

Fortunately, to use the workbook you do not need to know very much about branches. You really need to know only two things.

1. In the art-workbook repository, there is a branch named *develop*; this branch is the head of the project.
2. When the art-workbook team decides that a new stable version of the code is available, they `checkout` the develop branch and then start a new branch. This new branch, called a release branch has the same name as the version number of the release. For example, the code for version `v0_00_13` is found in branch `v0_00_13`, and so on. New work, towards the next release, continues on the develop branch.

This is a bit of simplification but it captures the big ideas. Users of art-workbook should always work with one of the release branches; and they should always consult the documentaion to learn which version of the code is matched to that version of the documentation.

Users of the art-workbook should never work in the develop branch; at any given time that branch may contain code that is still under development.

19.1.3 Seeing which Files you have Modified or Added

At any time you can check to see which files you have modified and which you have added. To do this, `cd` to your source directory and issue the `git status` command. Suppose that you have checked out version `v0_00_13`, modified `first.fcl` and added `second.fcl`. The `git status` command will produce the following output:

```
$ git status

# On branch v0_00_13
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
```

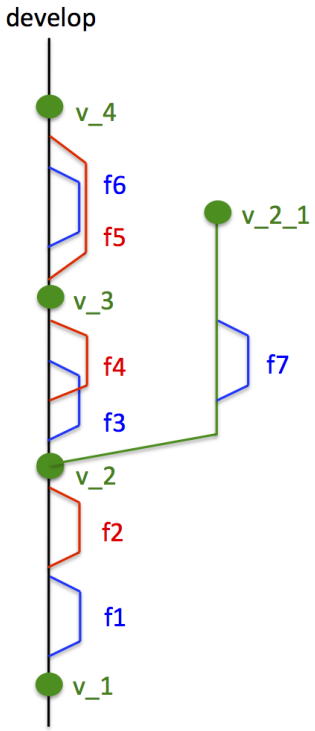


Figure 19.2: A figure to illustrate the idea of git branches, as used in the Workbook; the figure is described in the text.

```
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
# modified:   first.fcl
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# second.fcl
no changes added to commit (use "git add" and/or "git commit -a")
```

You should not issue the `git add` or `git commit` commands that are suggested above.

In the rare case that you have neither modified nor added any files, the output of `git status` will be:

```
$ git status
# On branch v0_00_13
```

20 *art* Run-time and Development Environments

20.1 The *art* Run-time Environment

Your *art* run-time environment consists of:

- your current working directory
- all of the directories that you can see and that contain relevant files, including system directories, project directories, product directories, and so on
- the files in the above directories
- the environment variables in your environment (not sure how to say this nicely)
- any aliases or shell functions that are defined

Figures 20.1, 20.2 and 20.3 show the elements of the run-time environment in various scenarios, and a general direction of information flow for job execution.

When you are running *art*, there are three environment variables that are particularly important:

- PATH
- LD_LIBRARY_PATH
- FHICL_FILE_PATH

They are colon-separated lists of directory names. When you type a command at the command prompt, or in a shell script, the (bash) shell splits the line using whitespace and the

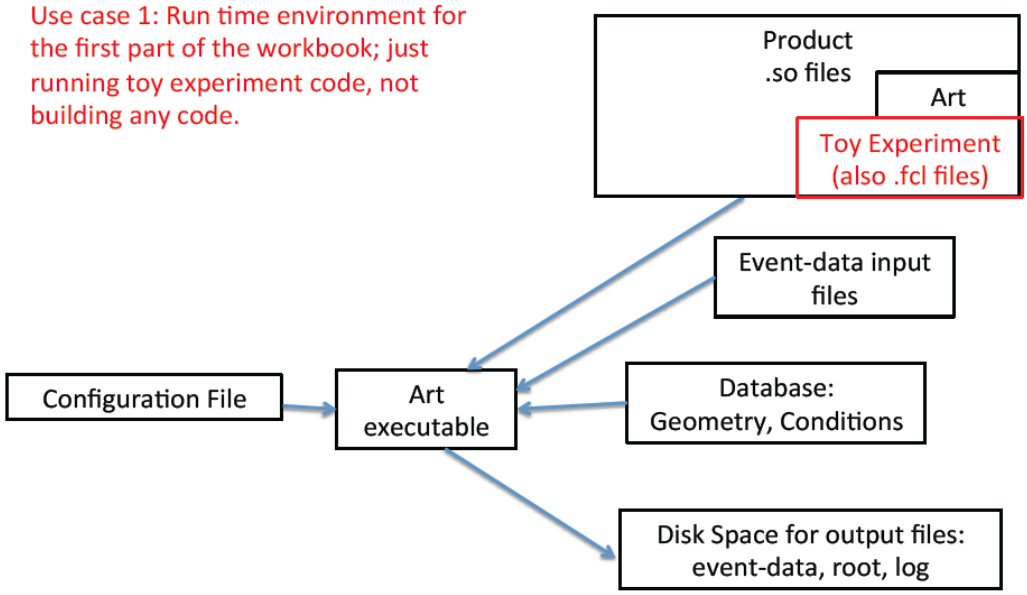


Figure 20.1: Elements of the *art* run-time environment, just for running the Toy Experiment code for the Workbook exercises

first element is taken as the name of a command. It looks in three places to figure out what you want it to do. In order of precedence:

1. it first looks at any aliases that are defined
2. secondly, it looks for shell keywords in your environment with the command name you provide
3. thirdly, it looks for shell functions in your environment with that name
4. then it looks for shell built-ins in your environment with that name
5. finally, it looks in the first directory defined in `PATH` and looks for a file with that name; if it does not find a match, it continues with the next directory, and so on, followed by the paths defined in the other two variables.

Some parts of the run-time environment will be established at login time by your login scripts. This is highly site-dependent. We will describe what happens at Fermilab - consult your site experts to find out if anything is provided for you at your remote site.

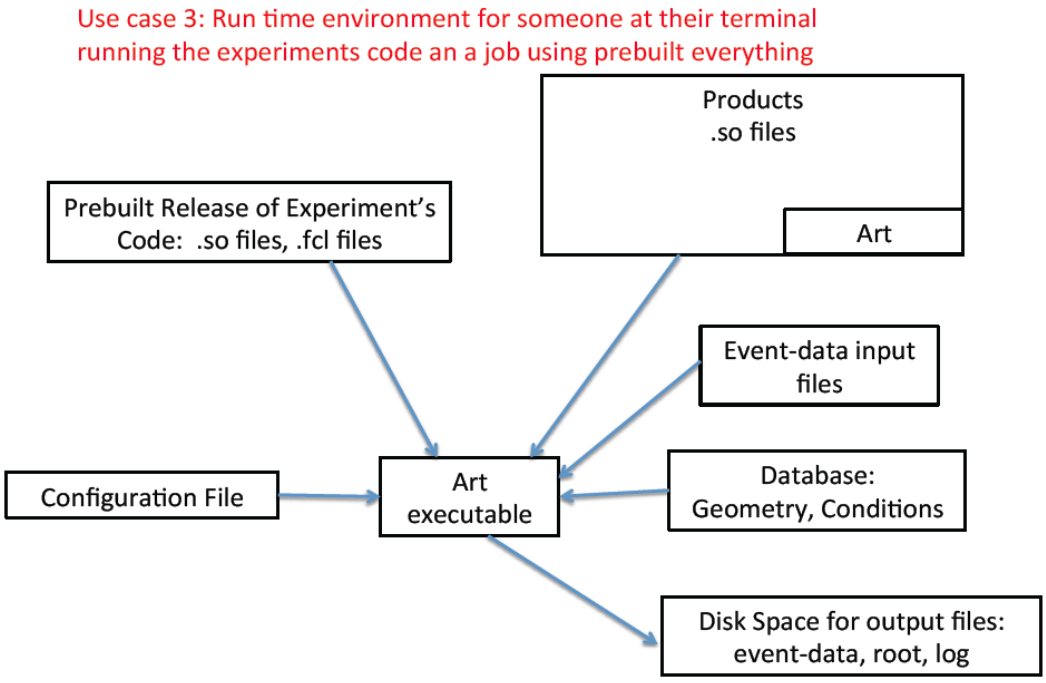


Figure 20.2: Elements of the *art* run-time environment for running an experiment's code (everything pre-built)

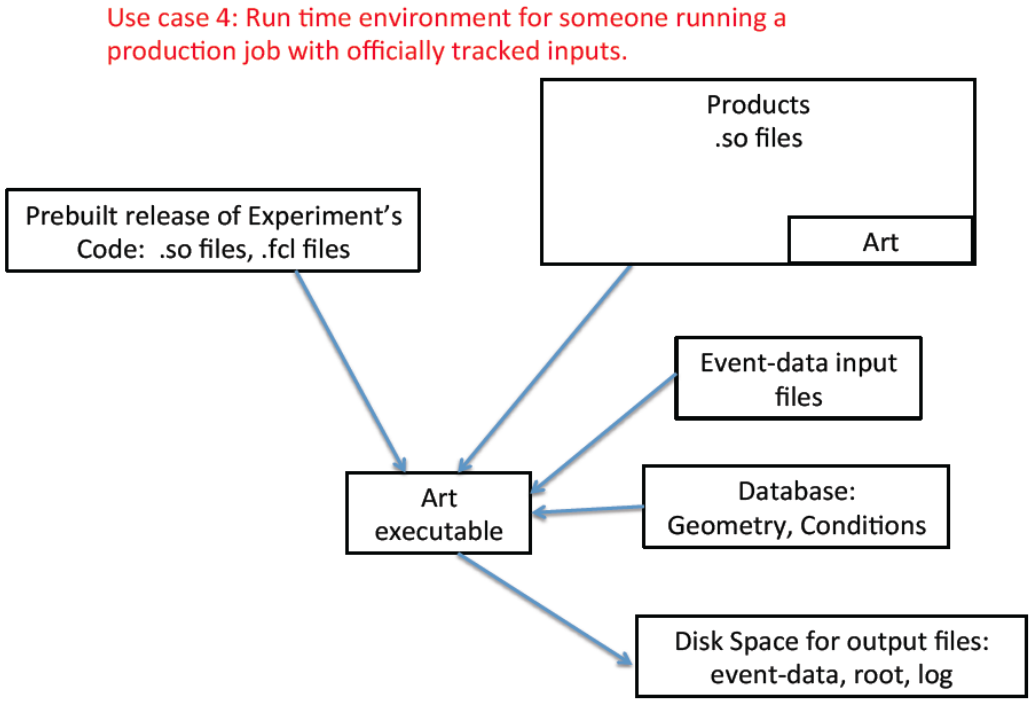


Figure 20.3: Elements of the *art* run-time environment for a production job with officially tracked inputs

When running the workbook, the interesting parts of your environment are established in two steps:

- source a site-specific setup script
- source a project-specific setup script

The Workbook, and the software suites for most IF experiments, are designed so that all site dependence is encoded in the site-specific setup script; that script adds information to your environment so that the project-specific scripts can be written to work properly on any site.

20.2 The *art* Development Environment

The development environment includes the run-time environment in Section 20.1 plus the following.

- the source code repository
- the build tools (these are the tools that know how to turn `.h` and `.cc` files in to `.so` files)
- additional environment variables and `PATH` elements that simplify the use of the above

Figures 20.4, 20.5 and 20.6 illustrate the development environment for various scenarios.

In some experiments the run-time and development environments are identical.

It turns out that there is no perfect solution for the job that build tools do. As a result, several different tools are widely used. Every tool has some pain associated with it. You never get to avoid pain entirely but you do get to pick where you will take your pain.

The workbook uses a build tool named **cetbuildtools**. Other projects have chosen `make`, `cmake`, `scons` and Software Release Tools (SRT). Here is something to watch out for: "build tools" written as two words refers generically to the above set of tools; but "buildtools" written as one word is the name of the executable that runs the build for **cetbuildtools**.

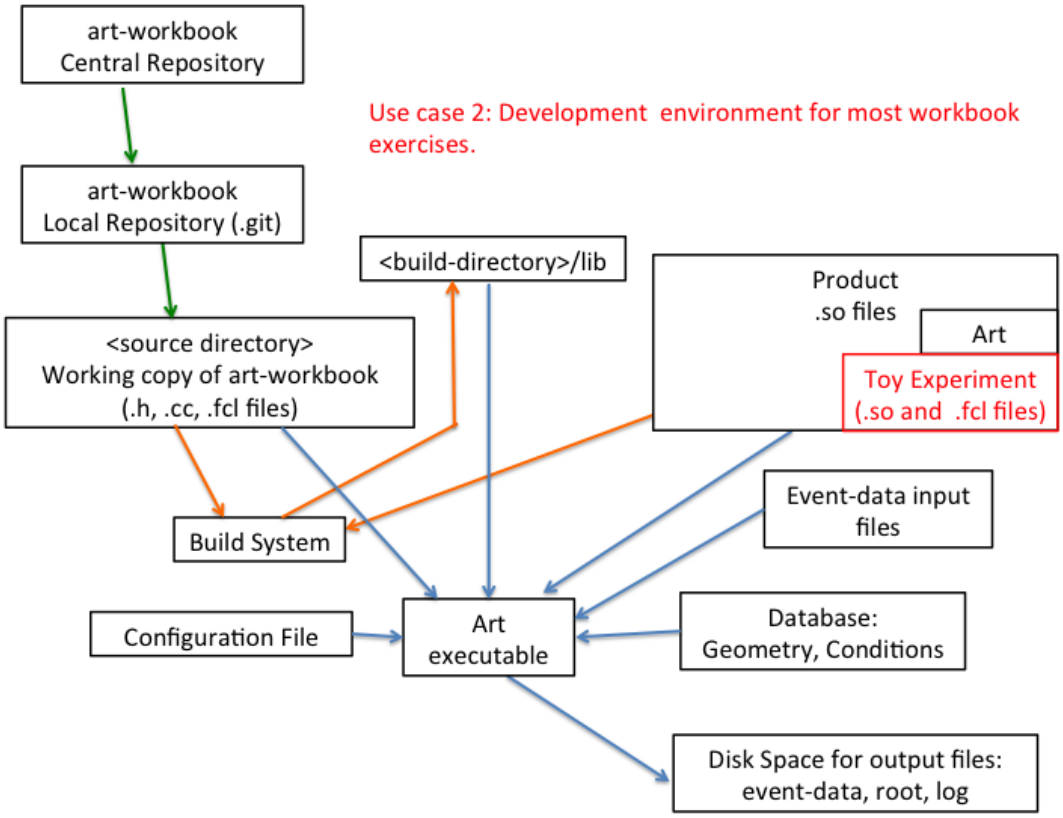


Figure 20.4: Elements of the *art* development environment as used in most of the Workbook exercises

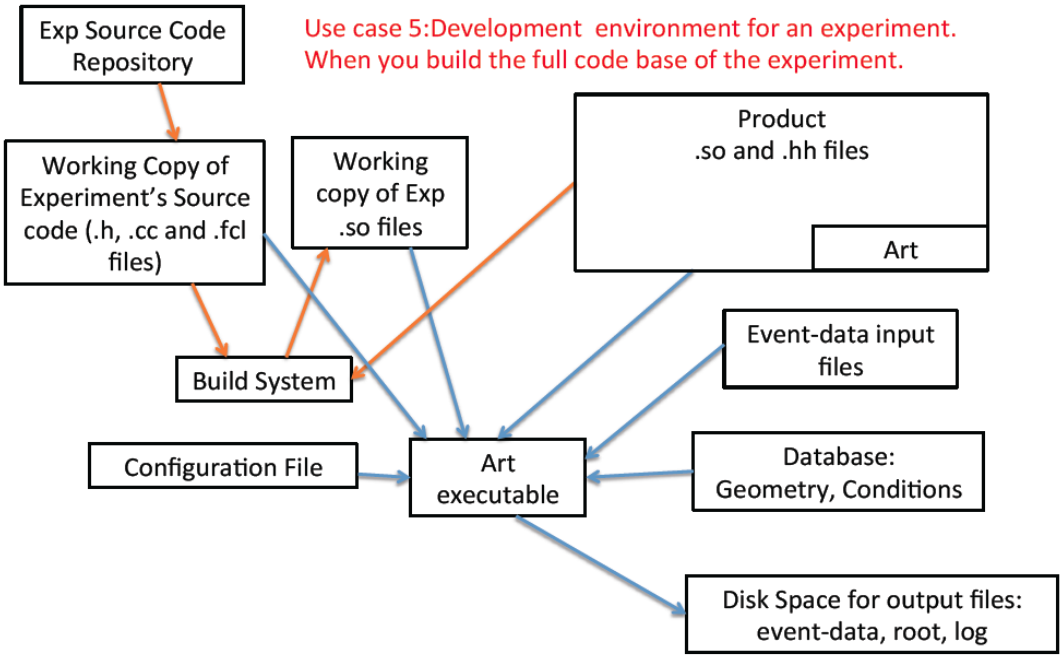


Figure 20.5: Elements of the *art* development environment for building the full code base of an experiment

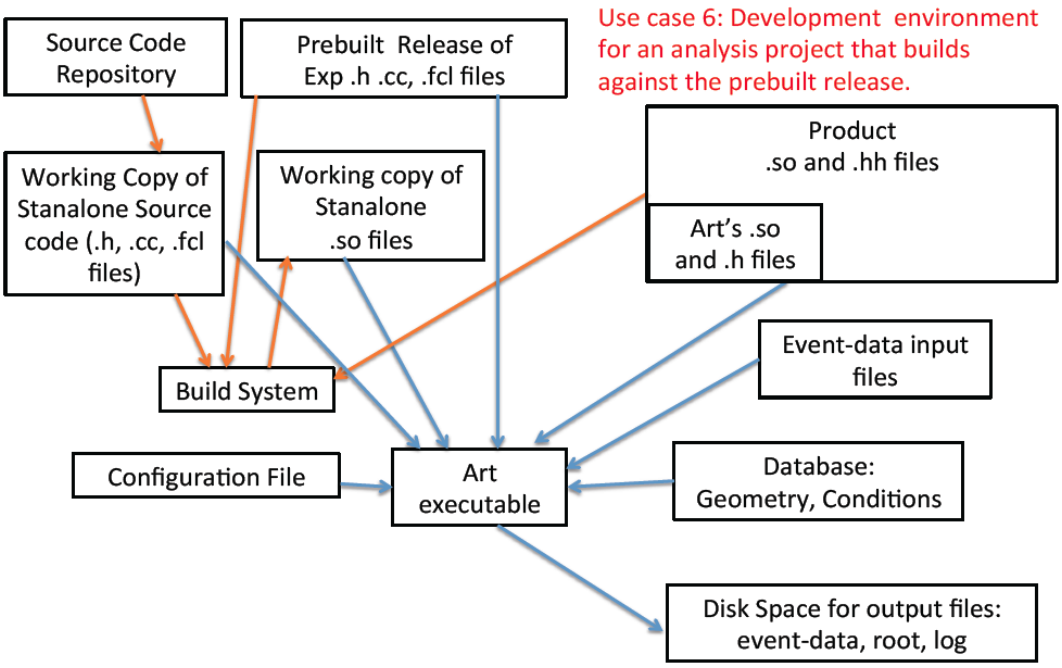


Figure 20.6: Elements of the *art* development environment for an analysis project that builds against prebuilt release

21 *art* Framework Parameters

This chapter describes all the parameters currently understood by the *art* framework, including by framework-provided services and modules. The parameters are organized by category (module, service or miscellaneous), and preceded by a general introduction to the expected overall structure of an *art* FHiCL configuration document.

21.1 Parameter Types

The parameters are described in tables for each module. The type of a defined parameter may be:

- TABLE: A nested parameter set, e.g., `set: { par1: 3 }`
- SEQUENCE: A homogeneous sequence of items,
  e.g., `list: [ 1, 1, 2, 3, 5, 8 ]`
- STRING: A string (enclosing double quotes not required when the string matches [A-Za-z_][A-Za-z0-9_]*). (Note: Identifiers when quoted do not function as special identifiers.) E.g.,

```
simpleString: g27
harderString: "a-1"
sneakystring1: "nil"
sneakystring2: "true"
sneakystring3: "false"
```

- COMPLEX: A complex number; e.g., `cnum: (3, 5)`
- NUMBER: A scalar (integer or floating point), e.g., `num: 2.79E-8`

- BOOL: A boolean, e.g.,

```
tbool: true
fbool: false
```

21.2 Structure of *art* Configuration Files

The expected structure of an *art* configuration file

Note, any parameter set is optional, although certain parameters or sets are expected to be in particular locations if defined.

```
# Prolog (as many as desired, but they must all be contiguous with onl
# whitespace or comments inbetween.
BEGIN_PROLOG
pset:
{
  nested_pset:
  {
    v1: [ a, b, "c-d" ]
    b1: false
    c1: 29
  }
}
END_PROLOG

# Defaulted if missing: you should define it in most cases.
process_name: PNAME

# Descriptions of service and general configuration.
services:
{
  # Parameter sets for known, built-in services here.
  # ...
```

```
  # Parameter sets for user-provided services here.
  user:
  {
  }

  # General configuration options here.
  scheduler:
  {
  }
}

# Define what you actually want to do here.
physics:
{
  # Parameter sets for modules inheriting from EDProducer.
  producers:
  {
    myProducer:
    {
      module_type: MyProducer
      nested_pset: @local::pset.nested_pset
    }
  }

  # Parameter sets for modules inheriting from EDFilter.
  filters:
  {
    myFilter: { module_type: SomeFilter }
  }

  # Parameter sets for modules inheriting from EDAnalyzer.
  analyzers:
  {
  }
```

```
  # Define parameters which are lists of names of module sets for
  # inclusion in end_paths and trigger_paths.

  p1: [ myProdroducer, myFilter ]
  e1: [ myAnalyzer, myOutput ]

  # Compulsory for now: will be computed automatically in a future
  # version of ART.

  trigger_paths: [ p1 ]
  end_paths: [ e1 ]
}

# The primary source of data: expects one and only one input source
parameter set.
source:
{
}

# Parameter sets for output modules should go here.
outputs:
{

}
```

21.3 Services

21.3.1 System Services

These services are always loaded regardless of whether a configuration is specified.

21.3.2 FloatingPointControl

These parameters control the behavior of floating point exceptions in different modules.

Table 21.1: *art* Floating Point Parameters

Enclosing Table Name	**Parameter Name**	**Type**	**Default**	**Notes**
services	floating_point_control	TABLE	{}	Top-level parameter set for the service
floating_point_control	setPrecisionDouble	BOOL	false	
	reportSettings	BOOL	false	
	moduleNames	SEQUENCE	[]	Each module name listed should also have its own parameter set within floating_point_control. One may also specify a module name of, "default" to provide default settings for the following items:
<module-name>	enableDivByZeroEx	BOOL	false	
	enableInvalidEx	BOOL	false	
	enableOverFlowEx	BOOL	false	
	enableUnderFlowEx	BOOL	false	

21.3.3 Message Parameters

These parameters configure the behavior of the message logger (this is a pseudo-service – not accessible via ServiceHandle).

Table 21.2: *art* Message Parameters

Enclosing Table Name	Parameter Name	Type	Default	Notes
services	message	TABLE		Top-level parameter set for the service
message				

21.3.4 Optional Services

These services are only loaded if a configuration is specified (although it may be empty).

21.3.5 Sources

21.3.6 Modules

Output modules

22 Job Configuration in *art*: FHiCL

Run-time configuration for *art* is written in the Fermilab Hierarchical Configuration Language (FHiCL, pronounced "fickle"), a language that was developed at Fermilab to support run-time configuration for several projects, including *art*. For this reason, this chapter will need to discuss FHiCL both as a standalone language and as used by *art*.

By convention, the names of FHiCL files end in `.fcl`. Job execution is performed by running *art* on a FHiCL configuration file, which is specified via an argument for the `-c` option:

```
$ art -c run-time-configuration-file.fcl
```

See Figure 20.1 in Section 20.1 to see how the configuration file fits into the run-time environment.

The FHiCL concept of *sequence*, as listed in brackets [], maps onto the C++ concept of std::vector, which is a sequence container representing an array that can change in size. Similarly, the FHiCL idea of table, as listed in curly brackets {}, maps onto the idea of fhicl::ParameterSet. . Note that ParameterSet is not part of *art*; it is part of a utility library *used* by *art*, FHICL-CPP, which is the C++ toolkit used to read FHiCL documents within *art*. FHiCL files provide the parameter sets to the C++ code, specified via module labels and paths, that is to be executed.

22.1 Basics of FHiCL Syntax

22.1.1 Specifying Names and Values

A FHiCL file contains a collection of definitions of the form

```
name : value
```

where “name” is a parameter that is assigned the value “value.” Many types of values are possible, from simple atomic values (a number, string, etc., with no internal whitespace) to highly structured table-like values; a value may also be a reference to a previously defined value. The white space on either side of the colon is optional. However, to include whitespace within a string, the string must be quoted (single or double quotes are equivalent in this case).

The fragment below will be used to illustrate some of the basics of FHiCL syntax:

```
# A comment.
// Also a comment.


name0  : 123            # A numeric value. Trailing comments
                        # work, too.
_name0  : 123            # Names can begin with underscores

name00 : "A quoted comment prefix, # or //, is just part of a
                                   # quoted string, not a comment"

name1:456.              # Another numeric value; whitespace is
                        # not important within a definition
name2 : -1.e-6
name3 : true            # A boolean value
NAME3 : false           # Other boolean value; names are case-
                        # sensitive.
name4 : red             # Simple strings need not be quoted
name5 : "a quoted string"
name6 : 'another quoted string'

name7 : 1 name8 : 2     # Two definitions on a line, separated by
                        # whitespace.
name9                   # Same as name9:3 ; newlines are just
:                       # whitespace,  which is not important.
```

```
 3


 namea : [ abc, def, ghi, 123 ]  # A sequence of atomic values.
                                 # FHiCL allows heterogeneous
                                 # sequences, which are not,
                                 # however, usable via the C++ API.

 nameb :                 # A table of definitions; tables may nest.
 {
    name0: 456
    name1: [7, 8, 9, 10 ]
    name2:
    {
      name0: 789
    }
 }

 namec : [ name0:{ a:1 b:2 } name1:{ a:3 c:4 } ]
                         # A sequence of tables.

 named : []              # An empty sequence
 namee : {}              # An empty table

 namef : nil             # An atomic value that is undefined.

abc : 1                  # If a definition is repeated twice within
abc : 2                  # the same scope, the second definition
def : [ 1, 2, 3 ]        #  will win (e.g., "abc" will be 2 and
def : [ 4, 5, 6 ]        # "def" will be [4,5,6])
name : {
  abc : 1
  abc : 2
}
```

```
cont1:{x: 1.0 y: 2.0 z: 3.0}  # Hierarchical (compound) names denote
cont1.x : 5                   # levels of scope; here set x in cont1 t
OR
cont2:[1, 2, 3]
cont2[0] : 1                  # Here, redefine the first (atomic) valu
                              # for cont2, assign it the value 1. I.e.
                              # no action.  Indices of PHiCL sequences
                              # begin with 0.  \fixme{right?}

name0:{ a:1 b:2 }
x : @local::name0.a   # Using reference notation "@local," this assign
                      #  to xthe value of a in table name0, in the
                      # line above, this value is 1.
```

22.1.2 FHiCL-reserved Characters and Identifiers

Several identifiers, characters and strings are *reserved to* FHiCL. What does this mean? Whenever FHiCL encounters a *reserved* string, FHiCL will interpret it according to the *reserved* meaning. Nothing prevents you from using these reserved strings in a name or value, but if you do, it is likely to confuse FHiCL. FHiCL may produce an error or warning, or it may silently do something different than what you intended. Bottom line: don't use reserved strings or symbols in the FHiCL environment for other than their intended uses.

The following characters, including the two-character sequence ::, are reserved to FHiCL:

```
, : :: @ [ ] { } ( )
```

The following strings have special meaning to FHiCL. They can be used as parameter values to pass to classes, e.g., to initialize a variable within a program, but their uses will not be fully described here because of subtleties and variations. As you work with C++ and FHiCL, the way to use them will become clearer.

true, false These values convert to a boolean

nil This value is associated with no data type. E.g. if `a : nil`, then `a` can't be converted

to any data type, and it must be redefined before use

infinity, +infinity, -infinity These values initialize a variable to positive (the first two) or negative (the third) infinity

BEGIN_PROLOG, END_PROLOG ()

The first six strings (three lines) above function as identifiers reserved to *art* only when entered as lower case and unquoted; the last two strings (the last line) are reserved to *art* only when they are in upper case, unquoted and at the start of a line. Otherwise these are just strings. You may include any of the above reserved characters and identifiers in a "quoted" string to prevent them from being recognized as reserved to *art*.

22.2 FHiCL Identifiers Reserved to *art*

FHiCL supports run-time configuration for several projects, not only for *art*. *art* reserves certain FHiCL names as identifiers that it uses in well-defined ways. (Other projects may use FHiCL names differently.) Within FHiCL files used by *art*, these FHiCL names obey scoping rules similar to C++. These identifiers appear in the FHiCL file with a scope, i.e.,

```
 identifier : {
...
}
```

if they define a list of modules or a processing block, or with square brackets

```
 identifier :[
...
]
```

if they define a list of paths.

The following is a list of the identifiers reserved to *art* and their meanings. In the outermost scope within a FHiCL file, the following can appear:

process_name A user-given name to identify the configuration defined by the FHiCL file (it is recommended to make it similar to the FHiCL file name). This must appear at the top of the file. It may not contain the underscore character (_).

source Identifies the data source, e.g., a file in ROOT format containing HEP events.

services Identifies ...

physics Identifies the block of code that configures the scientific work to be done on every event (as contrasted with the "bookkeeping" portions).

outputs List of output modules.

The following may appear within the `physics` scope:

producers Specifies the configurations of producer modules

analyzers Specifies the configuration of analyzer modules

filters Specifies the configuration of filter modules

trigger_paths Sequence of pathnames; the paths named here may contain only producer and/or filter modules.

end_paths Sequence of pathnames; the paths named here may contain only analyzer and output modules.

The last two elements specify which of the modules will be executed in an *art* job. (It is legal for a module to be configured but not to be executed.) To understand order of execution see Sections 22.4 and 22.7.

The identifier `process_name` is really only reserved to *art* within the outermost scope (but it would seem to be needlessly confusing to use `process_name` as the name of a parameter within some other scope). The names `trigger_paths` and `end_paths` are artifacts of the first use of the CMS framework, to simulate the several hundred parallel paths within the CMS trigger; their meaning should be come clear after reading the remainder of this page.

22.3 Structure of a FHiCL Run-time Configuration File for *art*

Here is a sample FHiCL file called `ex01.fcl` that will do a physics analysis using the code in the *art* module `Ex01_module.so` (the object file of the C++ source file

`Ex01_module.cc`). In this configuration, *art* will operate sequentially on the first three events contained in the source file `inputFiles/input01_data.root`.

```
#include "fcl/minimalMessageService.fcl"

process_name : ex01

source : {
  module_type : RootInput
  fileNames   : [ "inputFiles/input01_data.root" ]
  maxEvents   : 3
}

services : {
  message : @local::default_message
}

physics :{
  analyzers: {
    hello : {
      module_type : Ex01
    }
  }

  e1        : [ hello ]
  end_paths : [ e1 ]
}
```

Let's look at it step-by-step.

```
#include "fcl/minimalMessageService.fcl"
```

> Similar to C++ syntax, this effectively replaces the '#include' line with the contents of the named file. This particular file sets up a messaging service.

```
process_name : ex01
```

The value of the parameter `process_name` (`ex01`, here, the same as the FHiCL file name) identifies this *art* job. It is used as part of the identifier for data products produced in this job. For this reason, the value that you assign may not contain underscore (_) characters. If the `process_name` is absent, *art* substitutes a default value of "DUMMY."

```
source : {
  module_type : RootInput
  fileNames   : [ "inputFiles/input01_data.root" ]
  maxEvents   : 3
}
```

This `source` parameter describes where events come from. There may be at most one source module declared in an *art* configuration. At present there are two options for choosing a source module:

module_type : RootInput art::Events will be read from an input file or from a list of input files; files are specified by giving their pathname within the file system.

module_type : EmptyEvent Internally *art* will start the processing of each event by incrementing the event number and creating an empty art::Event. Subsequent modules then populate the art::Event. This is the normal procedure for generating simulated events.

Here RootInput is used; the data input file, in ROOT format, is assigned to the variable `fileNames`. The `maxEvents` parameter says: Look at only the first three events in this file. (A value of -1 here would mean "read them all.")

Note that if no source parameter set is present, *art* substitutes a default parameter set of:

```
source : {
  module_type : EmptyEvent
  maxEvents : 1
}
```

See the web page about configuring input and output modules for details about what other parameters may be supplied to these parameter sets.

```
services : {
  message : @local::default_message
}
```

Before starting processing, this puts the message logger in the recommended configuration.

```
physics :{
  analyzers: {
    hello : {
      module_type : Ex01
    }
  }
```

In *art*, `physics` is the label for a portion of the run-time configuration of a job. It contains the "meat" of the configuration, i.e., the scientific processing instructions, in contrast to the more administrative or bookkeeping information. The `physics` block of code may contain up to five sections, each labeled with a reserved identifier (that together form a parameter set within the FHiCL language); the strings are *analyzers*, *producers*, *filters*, *trigger_paths* and *end_paths*. In our example it's set to `analyzers`.

The `analyzers` identifier takes values that are FHiCL tables of parameter sets (this is true also for `filters` and `producers`). Here it takes the value `hello`, which is defined as a table with one parameter, namely `module_type`, set to the value `Ex01`. The setup defined a variable called LD_LIBRARY_PATH; *art* knows to match the value defined by the name `module_type` to a C++ object file with the name `Ex01_module.so` somewhere in the path defined by LD_LIBRARY_PATH.

We will expand on the `physics` portion of the FHiCL configuration in Section 22.5.

```
e1        : [ hello ]
end_paths : [ e1 ]
```

22.4 Order of Elements in a FHiCL Run-time Configuration File for *art*

In FHiCL files there are very, very few places in which order is important. Here are the places where it matters:

- A `#include` must come before lines that use names found inside the `#include`.
- A later definition of a name overrides an earlier definition of the same name.
- The definition of a name resolved using `@local` needs to be earlier in the file than the place(s) where it is used.

- Within a trigger path, the order of module labels is important.

Here is a list of *a few places* (of many) where order does not matter. This list is by no means exhaustive.

- Inside the physics scope, the order in which modules are defined does NOT matter for filters and analyzers blocks. These blocks define the run-time configurations of *instances* of modules.
- The five *art*-reserved words that appear in the outermost scope of a FHiCL file can be in any order. You could put outputs first and process_name last, as far as FHiCL cares. It may be more difficult for humans to follow, however.
- Within the services block, the services may appear in any order.

Regarding trigger_paths and end_paths, the following is a conceptual description of how *art* processes the FHiCL file:

1. *art* looks at the trigger_paths sequence. It expands each trigger path in the sequence, removes duplicate entries and turns the result into an ordered list of module labels. The final list has to obey the order of each contributing trigger path, but there are no other ordering constraints.
2. It does the same for the end_paths sequence but there is no constraint on order.
3. It makes one big sequence that contains everything in 1 followed by everything in 2.

4. It looks throughout the file to find parameter sets to match to each module label in the big list in 3.

5. It gives warning messages if there are left over parameter set definitions not matched to any module label in 3.

6. It then parses the rest of the physics block to make a "dictionary" that matches module labels to their configuration.

A conceptual description for the porcessing of services is as follows:

1. *art* first makes a list of all services, sorted alphabetically.

2. It makes a dictionary that matches service names to their parameter sets. A collorary is that service names must be unique within an *art* job.

3. *art* has some "magic" services that it knows about internally. It loads the `.so` file for each of them and constructs the services.

4. It loads the `.so` files for all of the services and calls their constructors, passing each service its proper parameter set.

5. It works through its list of modules in 5 - it loads the `.so` and calls the constructor, passing the constructor the right parameter set.

6. It gives warning messages if there are left-over parameter set definitions not matched to any module label in 3.

When one service relies on another, things get a bit more complicated. If service A requires that service B be constructed first, then the constructor of service A must ask *art* for a handle to service B. When this happens, *art* will start to construct service A since it is alphabetically first. When the constructor of A asks for a handle to B, *art* will interupt the construction of service A, construct service B, and return to finish service A. Next, *art* will see that the next thing in the list is B, but it will notice that B has already been constructed and will skip to the next one.

Got that? Whew!

22.5 The *physics* Portion of the FHiCL Configuration

art looks for the experiment code in *art* modules. These must be referenced in the FHiCL file via *module labels*, which are just variable names that take particular values, as this section will describe. The structure of the FHiCL file – or a portion thereof – therefore defines the event loop for *art* to execute. The event loop, as defined in the FHiCL file, is collected into a scope labeled *physics*.

For a module label you may choose any name, as long as it is unique within a job, contains no underscore (_) characters and is not one of the names reserved to *art*. In the sample physics scope code below, we define `aProducer`, `bProducer`, `checkAll`, `selectMode0` and `selectMode1` as module labels.

```
physics: {

 producers : {
    aProducer: { module_type: MakeA }
    bProducer: { module_type: MakeB }
  }

 analyzers : {
    checkAll: { module_type: CheckAll }
  }

 filter : {
    selectMode0: {
      module_type: Filter1
      mode: 0
    }
    selectMode1: {
      module_type: Filter1
      mode: 1
    }
  }
```

The minimum configuration of a module is:

```
<moduleLabel> : { module_type : <ClassName> }
```

for example, in our code above:

```
aProducer: { module_type: MakeA }
```

`aProducer` is the module label and `MakeA` corresponds to a module of experiment code (i.e., an *art* module) named `MakeA_module.so`, which in turn was built from `MakeA_module.cc`. Since it falls within the scope `producers`, it must be a module of type EDProducer.

Let's take this a step farther, and assume that this EDProducer-type module MakeA accepts four arguments that we want to provide to *art*. The configuration may look like this:

```
moduleLabel : {
   module_type : MakeA
   pname0 : 1234.
   pname1 :  [ abc, def]
   pname2 : {
        name0: {}
        }
}
```

This list under `module_type : MakeA` represents parameters that will be formed into a `fhicl::ParameterSet` object and passed to the module `MakeA` as an argument in its constructor. `pname0` is a double, `pname1` is a sequence of two atomic character values, `pname2` consists of a single table named `name0` with undefined contents.

Note that *paths* are lists of module labels, while the two reserved names, `trigger_paths` and `end_paths` are lists of paths.

22.6 Choosing and Using Module Labels and Path Names

For a module label or a path name, you may choose any name so long as it is unique within a job, contains no underscore (_) characters and is not one of the names reserved to *art*

(see Section 22.2.

Any name that is a top-level name inside of the `physics` parameter set is either a reserved name or the name of a path.

It is important to recognize which identifiers are module labels and which are path names in a FHiCL file. It is also important to distinguish between a class that is a module and instances of that module class, each uniquely identified by a module label.

art has several rules that were recommended practices in the old framework but which were not strictly enforced by that framework. *art* enforces some of these rules and will, soon, enforce all of them:

- A path may go into either the `trigger_paths` list or into the `end_paths` list, but not both.
- A path that is in the `trigger_paths` list may only contain the module labels of producer modules and filter modules.
- A path that is in the `end_paths` list may only contain the module labels of analyzer modules and output modules.

Analyzer modules and the output modules may be separated into different paths; that might be convenient at some times but it is not necessary. On the other hand, keeping trigger paths separate has real meaning.

22.7 Scheduling Strategy in *art*

A set of scheduling rules is enforced in *art*. (Some of the details are remnants of compromises and conflicting interests with CMS.) One of the top-level rules in the scheduler is that all producers and filters must be run first, using the ordering rules specified below. After that, all analyzer and output modules will be run. Recall that analyzer modules and output modules may not modify the event, nor may they produce side effects that influence the behavior of other analyzer or output modules. Therefore, *art* is free to run analyzer and output modules in any order.

The full description of the scheduler strategy is given below:

- If a module name appears in the definition of a path name but it is not found among the the list of defined module labels, FHiCL will issue an error.
- One each event, before executing any of the paths, execute the source module.
- On each event, execute all of the paths listed in the trigger_paths.
  - Within one path, the order of modules listed in the path is followed strictly; at present there is one exception to this: see the discussion about the remaining issues
  - *art* can identify module labels that are in common to several trigger_paths and will execute them only once per event. In the above example, aProducer and bProducer are executed only once per event.
  - The various paths within the trigger_paths may be executed in any order, subject to the above constraints.
  - If a path contains a filter, and if the filter return false, then the remainder of the path is skipped.
  - The module name of a filter can be negated in path using, !moduleLabel; in this case the path will continue if the filter returns false and will be aborted if the filter returns true.
  - If the module label of a filter appears in two paths, negated in one path and not negated in the other, art will only run the instance of filter module once and will use the result in both places.
  - If a module in a trigger path throws, the default behaviour of art is to stop all processing and to shut down the job as gracefully as possible. Art can be configured, at run time, so that, for selected exceptions, it behaves differently. For example it can be configured to continue with the current trigger path, skip to the next trigger path, skip to the next event, and so on.
- On each event, execute all of the paths listed in the end_paths.
  - The module labels listed in end_paths are executed exactly once per event, regardless of how many paths there are in the trigger_paths and regardless of any filters that failed.

- If a module label appears multiple times among the end paths, it is executed only once. No warning message is given.
- Even if all trigger_paths have filters that fail, all module labels in the end path will be run.
- End_path is free to execute the modules in the end_path in any order.
- If a module in the end_path throws, the default response of art is to make a best effort to complete all other modules in the end path and then to shutdown the job in an orderly fashion. This behaviour can be changed at run-time by adding the appropriate parameter set to the top level .fcl file.

◦ One can ask that an output module be run only for events that pass a given trigger_path; this is done using the SelectEvents parameter set,

◦ At present there is no syntax to ask that an analyzer module be run only for events that pass or fail some of the trigger paths. A planned improvement to *art* is to give analyzer modules a SelectEvents parameter that behaves as it does for output modules.

◦ If a path appears in neither the trigger_paths nor the end_paths, there is no warning given.

◦ If a module label appears in no path, a warning will be given.

In the above there is a lot of focus on which groups of modules are free to be run in an arbitrary order. This is laying the groundwork for module-parallel execution: *art* is capable of identifying which modules may be run in parallel and, on a multi-core machine, art could start separate threads for each module. At present both ROOT and G4 are not thread-safe so this is not of immediate interest. But there are efforts underway to make both of these thread-safe and we may one day care about module-parallel execution; our interest in this will depend a great deal on the future evolution of the relative costs of memory and CPU.

For simple cases, in which there is one trigger path with only a few modules in the path, and one end path with only a few modules in the path, the extra level of bookkeeping is just extra typing with no obvious benefit. The benefit comes when many work groups wish to run their modules on the same events during one art job; perhaps this is a job skimming off

many different calibration samples or perhaps it is a job selecting many different streams of interesting Monte Carlo events. In such a case, each work group needs only to define their own trigger path and their own end path, without regard for the requirements of other work groups; each work group also needs to ensure that their paths are added to the end_paths and trigger_paths variables. Art will then automatically, and correctly, schedule the work without redoing any work twice and without skipping work that must be done. This feature came for free with art and, while it imposes a small burden for novice users doing simple jobs, it provides an enormously powerful feature for advanced users. Therefore it was retained in art when some other features were removed.

22.8 Scheduled Reconstruction using Trigger Paths

Consider the following problem. You wish to run a job that has:

- Two producers MakeA_module.cc and MakeB_module.cc. You want to run both producers on all events.
- One analyzer module that you want to run on all events, CheckAll_module.cc.
- You have a filter module, Filter1_module.cc that has two modes, 0 and 1; the mode can be selected at run time via the parameter set.
- You wish to write all events that pass mode 0 of the filter to the file file0.root and you wish to write all events that pass mode 1 of the filter to file1.root

Here is code that would accomplish this:

```
process_name: filter1

source: {
   # Configure some source here.
}

physics: {

 producers : {
    aProducer: { module_type: MakeA }
```

```
    bProducer: { module_type: MakeB }
  }

 analyzers : {
    checkAll: { module_type: CheckAll }
  }

 filter : {
    selectMode0: {
      module_type: Filter1
      mode: 0
    }
    selectMode1: {
      module_type: Filter1
      mode: 1
    }
  }

  mode0: [ aProducer, bProducer, selectMode0 ]
  mode1: [ aProducer, bProducer, selectMode1 ]
  analyzermods: [ checkAll  ]
  outputFiles:  [ out0, out1 ]

  trigger_paths : [ mode0, mode1 ]
  end_paths : [ analyzermods, outputFiles ]
}

outputs: {
  out0: {
   module_type: RootOutput
   fileName: "file0.root"
   SelectEvents: { SelectEvents: [ mode0 ] }
  }

  out1: {
```

```
   module_type: RootOutput
   fileName: "file1.root"
   SelectEvents: { SelectEvents: [ mode1 ] }
  }

}
```

Recall that the names process_name, source, physics, producers, analyzers, filters, trigger_paths, end_paths and outputs are reserved to *art*. The names aProducer, bProducer, checkAll, selectMode0, selectMode1, out0 and out1 are module labels, and these are names of paths: mode0, mode1, outputFiles, analyzermods.

22.9 Reconstruction On-Demand

22.10 Bits and Pieces

What variables are known to *art*? physics (which has the five reserved identifiers: filters, analyzers, producers, trigger paths and end paths), what else? input file type RootInput

I know that trigger path are // different from end paths, they can contain different types of modules; // event gets frozen after trigger path.

art knows to match the value defi

ned by the name 'module_name" to a C++ object fi

le with the name module_name_module.so" somewhere in the path defi

ned by LD LIBRARY PATH.

Further information on the FHiCL language and usage can be found at the mu2e FHiCL page.

Part IV

Appendices

A CLHEP

A.1 Introduction

The wikipedia entry for CLHEP,http://en.wikipedia.org/wiki/CLHEP, describes it as:

> CLHEP (short for A Class Library for High Energy Physics) is a C++ library that provides utility classes for general numerical programming, vector arithmetic, geometry, pseudorandom number generation, and linear algebra, specifically targeted for high energy physics simulation and analysis software. The project is hosted by CERN and currently managed by a collaboration of researchers from CERN and other physics research laboratories and academic institutions. According to the project's website, CLHEP is in maintenance mode (accepting bug fixes but no further development is expected).

The *art* workbook uses CLHEP, as do many of the experiments that use *art*. In both the *art* run-time environment and the *art* development environment CLHEP is made available via UPS and is rooted at $CLHEP_DIR. The CLHEP header files can be found at $CLHEP_INC and the libraries can be found at $CLHEP_LIB_DIR. These enviroment variables will also be defined in the corresponding environments for your experiment.

This appendix will discuss those parts of CLHEP that are important for the *art* workbook and will fill in some background information that is assumed by the CLHEP documentation but is not explicitly stated elsewhere.

CLHEP is divided in packages and the *art* workbook uses classes from four of these packages:

Matrix Support for linear algebra.

Random Support for random engines and random distributions. The distinction between these two ideas will be discussed in *a section to be written in the future*.

Units Support for a standard set of units and for transformations among different units. It also provides the values of many physical constants.

Vector Support for 2-vectors, 3-vectors, 4-vectors.

A.1.1 Multiple Meanings of Vector

CLHEP uses the word vector in two different senses both of which are different from the use of the word in the standard library template `std::vector`. The Matrix package supports linear algebra, by providing classes to represent matrices and vectors of arbitrary dimensions; the package supports operations such as matrix multipilcation and the computation of the inverse, transpose, determinant and trace of a matrix. The Vector package, on the other hand, provides classes that represent a point on a plane, a point in 3-space or a point in 4-dimensional space-time; the package supports operations such dot products, cross products, rotations and Lorentz transformations.

The Vector package does not make a distinction between positions, displacements, velocities and momentum. The same classes are used for all four.

A.2 CLHEP Documentation

The CLHEP home page is http://proj-clhep.web.cern.ch/proj-clhep.

The following is a direct link to the CLHEP documentation page:
http://proj-clhep.web.cern.ch/proj-clhep/index.html#docu

In many cases the documentation for CLHEP is simply the code or the comments in the code. You can view the header files by looking under $CLHEP_INC. You can view the source files by looking under $CLHEP_DIR/source. A more convenient format to view the header files is to use the CLHEP Doxygen site:
http://proj-clhep.web.cern.ch/proj-clhep/doc/CLHEP_2_1_3_1/doxygen/html/
Doxygen simply presents the information found in the header file in a format that is easier to view than the header file itself.

To get information about a CLHEP class, go to the Doxygen page, click on the tab named "Classes", and use your browswer's search function to find the name the class.

A.3 CLHEP Header Files

The syntax to include a CLHEP header file is:

```
#include "CLHEP/<package-name>/<filename>.h"
```

where `package-name` is the name of the CLHEP package to which the header file belongs and where `filename` is filename of the header file.

Almost all of the class names in CLHEP begin with the prefix `HEP`, for example `LorentzVector`, which is the CLHEP representation of a 4-vector. The handful of exceptions to this rule are helper classes used internally by CLHEP.

In most cases, the name of the header file for a class is the name of the class, excluding the leading `HEP`. For example, the header file for `HepLorentzVector` is `CLHEP/Vector/LorentzVector.h`. Some header files also contain the declarations of helper classes that are used by the main class. A few header files contain the declarations of several related classes.

There is one important header file that follows an unusual naming pattern. The header file `CLHEP/Vector/ThreeVector.h` declares the classes `Hep2Vector` and `Hep3Vector` that describe a point in a plane and a point in 3-space, respectively.

A.3.1 .icc Files

The CLHEP package was developed in the mid 1990's and the authors of CLHEP chose to use a convention that was current at that time. CLHEP header files contain only declarations. When inline implemenations are required, CLHEP puts the these in a file named the same as the header file but with `.h` replacd with `.icc`. The `.icc` file is included near the end of the `.h` file. For example the inline implementation for `Hep3Vector` is found in

```
#include "CLHEP/Vector/ThreeVector.icc"
```

The pattern of using `.icc` files to segregate inline implementation from declarations is no longer recommended but CLHEP retains it for backwards compatibility.

A.4 CLHEP Namespace

All identifiers defined by CLHEP are in the CLHEP namespace.

A.4.1 using Declarations and Directives

An example of a *using directive* is:

```
1 using namespace CLHEP;
```

An example of a *using declaration* is:

```
1 using CLHEP::Hep3Vector;
```

Remember that you must never code using directives or using declarations in header files. You should only use them in source files. .

Even in source code files, never write:

```
1 using namespace CLHEP;
```

Why? The CLHEP Units package defines many identifiers with commonly used short names, `m`, `g` and `s`; in addition there are 17 two character identifiers and 24 three character identifiers, such as `mm`, `m2`, `deg`, `cm3`, `amu`, and so on. Many of these short identifiers are commonly used in code, `m` for mass, `s` for an arc length and so on. If you give a using directive for the namespace CLHEP then all of these short names will be defined with the scope of your code.

A common programming error is to forget to declare a variable before using it. Normally the compiler will recognize this error and issue a diagnostic message. If, on the other hand, one of your undeclared variables matches one of the CLHEP variable names, and if you have used `using namespace CLHEP`, then the compiler will not recognize the error and will not issue a diagnostic. You will need to find the error by tedious debugging.

It is acceptable to code using declarations for individual identifiers.

```
1 using CLHEP::Hep3Vector;
```

but only if the identifier, stripped of the `CLHEP::`, is unambiguous to the reader and is not a commonly used variable name.

A.5 The Vector Package

The *art* workbook uses the following classes from the Vector package:

`Hep3Vector` A vector in 3-space.

`HepLorentzVector` A 4-vector in 4-dimensional space time.

`HepBoost` Lorentz boosts from one inertial frame to another; it operates on objects of type `HepLorentzVector`.

A.6 The Matrix Package

The *art* workbook uses the following classes from the Matrix package:

`HepMatrix` A general $n \times m$ matrix class.

`HepSymMatrix` A class that represents symmetric matrices.

`HepVector` A column-vector ($n \times 1$ matrix) class.

A.7 The Random Package

The *art* workbook uses the following classes from the Random package:

`HepRandomEngine` The base class from which all CLHEP random engines must inherit.

`HepJamesRandom` A random engine that implements an algorithm described by F. James of CERN.

`RandFlat` A distribution that returns a random variate that is flat on a specified domain.

`RandGaussQ` A distribution that returns a random variate that is distributed as a Gaussian distribution.

`RandPoissonQ` A distrbution that returns a random variate that is distributed as a Poisson distribution.

The two classes with names ending in `Q` have no internal state except for the state of the underlying engine.

Part V

Index

Index